

## 版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。





# 机器学习之路

Caffe、Keras、scikit-learn 实战

阿布 胥嘉幸 编著

绕过理论障碍，理解机器学习，  
打通一条由浅入深的机器学习之路。  
丰富的实战案例讲解，  
介绍如何将机器学习技术运用到  
股票量化交易、图片渲染等领域。



中国工信出版集团



电子工业出版社  
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
<http://www.phei.com.cn>

## 作者简介

---

### 阿布

多年互联网金融技术从业经验，曾就职于奇虎360、百度互联网证券、百度金融等互联网型金融公司，现自由职业，个人量化交易者，擅长个人中小资金量化交易领域系统开发，以及为中小型量化私募基金提供技术解决方案、技术支持、量化培训等工作。

### 胥嘉幸

北京大学硕士，先后就职于百度金融证券、百度糯米搜索部门。多年致力于大数据机器学习方面的研究，有深厚的数学功底和理论支撑。在将机器学习技术融于传统金融量化领域方面颇有研究。



# 机器学习之路

Caffe、Keras、scikit-learn 实战

阿布 胥嘉幸 编著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

## 内 容 简 介

机器学习需要一条脱离过高理论门槛的入门之路。

本书《机器学习篇》从小红帽采蘑菇的故事开篇，介绍了基础的机器学习分类模型的训练（第1章）。如何评估、调试模型？如何合理地发掘事物的特征？如何利用几个模型共同发挥作用？后续章节一步一步讲述了如何优化模型，更好地完成分类预测任务（第2章），并且初步尝试将这些技术运用到金融股票交易中（第3章）。

自然界最好的非线性模型莫过于人类的大脑。《深度学习篇》从介绍并对比一些常见的深度学习框架开始（第4章），讲解了DNN模型的直观原理，尝试给出一些简单的生物学解释，完成简单的图片识别任务（第5章）。后续章节在此基础上，完成更为复杂的图片识别CNN模型（第6章）。接着，本书展示了使用Caffe完成一个完整的图片识别项目，从准备数据集，到完成识别任务（第7章）。后面简单描述了RNN模型（第8章），接着展示了一个将深度学习技术落地到图片处理领域的项目（第9章）。

本书适合能看懂Python代码，对机器学习感兴趣，期望入门的读者。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

### 图书在版编目(CIP)数据

机器学习之路: Caffe、Keras、scikit-learn 实战 / 阿布, 胥嘉幸编著. —北京: 电子工业出版社, 2017.8  
ISBN 978-7-121-32160-3

I. ①机… II. ①阿… ②胥… III. ①机器学习 IV. ①TP181

中国版本图书馆CIP数据核字(2017)第165543号

责任编辑: 安娜

印 刷: 三河市良远印务有限公司

装 订: 三河市良远印务有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路173信箱 邮编100036

开 本: 787×980 1/16 印张: 20.5 字数: 405千字

版 次: 2017年8月第1版

印 次: 2017年8月第1次印刷

定 价: 79.00元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888, 88258888。

质量投诉请发邮件至 [zltz@phei.com.cn](mailto:zltz@phei.com.cn), 盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

本书咨询联系方式: 010-51260888-819, [faq@phei.com.cn](mailto:faq@phei.com.cn)。

# 前言

越来越多的人期待能挤进机器学习这一行业，这些人往往有一些编程和自学能力，但数学等基础理论能力不足。对于这些人群，从头开始学习概率统计等基础学科是痛苦的，如果直接上手使用机器学习工具往往又感到理解不足，缺少点什么。本书就是面向这一人群，避过数学推导等复杂的理论推衍，介绍模型背后的一些简单直观的理解，以及如何上手使用。本书希望能够得到这些人的喜爱。

本书包含两部分：机器学习篇和深度学习篇。

机器学习篇（1~3 章）主要从零开始，介绍什么是数据特征，什么是机器学习模型，如何训练模型、调试模型，以及如何评估模型的成绩。通过一些简单的任务例子，讲解在使用模型时如何分析并处理任务数据的特征，如何组合多个模型共同完成任务，并在第 3 章初步尝试将机器学习技术运用到股票交易中，重复熟悉这些技术的同时，感受机器学习技术在落地到专业领域时常犯的错误。

深度学习篇（4~9 章）则主要介绍了一些很基础的深度学习模型，如 DNN、CNN 等，简单涵盖了一些 RNN 的概念描述。我们更关注模型的直观原理和背后的生物学设计理念，希望读者能够带着这些理解，直接上手应用深度学习框架。

说一点关于阅读本书的建议。本书在编写时不关注模型技术的数学推导及严谨表述，转而关注其背后的直观原理理解。建议读者以互动执行代码的方式学习，所有示例使用 IPython Notebook 编写。读者可在 Git 上找到对应章节的内容，一步一步运行书中讲解的知识点，直观感受每一步的执行效果。具体代码下载地址：<https://github.com/bbfamily/abu>。

本书适合有 Python 编程能力的读者。如果读者有简单的数学基础，了解概率、矩阵则更佳。使用过 Numpy、pandas 等数据处理工具的读者读起来也会更轻松，但这些都不是必需的。如果读者缺乏 Python 编程能力，或者希望进一步获得 Numpy、pandas 等工具

使用相关的知识，可以关注公众号：abu\_quant，获得一些技术资料及文章。

感谢出版社提供机会让我们编写本书，感谢编辑不辞辛苦地和我沟通排版等细节问题。

本书的完成同样需要感谢我们的几位朋友：吴汶（老虎美股）、刘兆丹（百度金融），感谢你们在本书编写作过程中提供的有力支持。感谢本书的试读人员：蔡志威、李寅龙。

---

轻松注册成为博文视点社区用户（[www.broadview.com.cn](http://www.broadview.com.cn)），扫码直达本书页面。

- **提交勘误：**您对书中内容的修改意见可在 [提交勘误](#) 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方 [读者评论](#) 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/32160>



# 目录

## 第一篇 机器学习篇

第 1 章 初识机器学习	2
1.1 机器学习——赋予机器“学习”的灵魂	2
1.1.1 小红帽识别毒蘑菇	2
1.1.2 三种机器学习问题	6
1.1.3 常用符号	6
1.1.4 回顾	7
1.2 KNN——相似的邻居请投票	7
1.2.1 模型原理	7
1.2.2 鸢尾花卉数据集 (IRIS)	9
1.2.3 训练模型	9
1.2.4 评估模型	12
1.2.5 关于 KNN	14
1.2.6 运用 KNN 模型	15
1.2.7 回顾	16
1.3 逻辑分类 I: 线性分类模型	16
1.3.1 参数化的模型	16
1.3.2 逻辑分类: 预测	18
1.3.3 逻辑分类: 评估	22
1.3.4 逻辑分类: 训练	23
1.3.5 回顾	24
1.4 逻辑分类 II: 线性分类模型	24
1.4.1 寻找模型的权重	24



1.4.2	去均值和归一化	31
1.4.3	实现	33
1.4.4	回顾	34
<b>第2章</b>	<b>机器学习进阶</b>	<b>35</b>
2.1	特征工程	35
2.1.1	泰坦尼克号生存预测	35
2.1.2	两类特征	38
2.1.3	构造非线性特征	41
2.1.4	回顾	45
2.2	调试模型	46
2.2.1	模型调试的目标	46
2.2.2	调试模型	49
2.2.3	回顾	52
2.3	分类模型评估指标	53
2.3.1	混淆矩阵系指标	53
2.3.2	评估曲线	58
2.3.3	回顾	61
2.4	回归模型	61
2.4.1	回归与分类	61
2.4.2	线性回归	62
2.4.3	波士顿房价预测	66
2.4.4	泰坦尼克号生存预测：回归预测特征年龄 Age	69
2.4.5	线性模型与非线性模型	72
2.4.6	回顾	73
2.5	决策树模型	73
2.5.1	信息与编码	74
2.5.2	决策树	76
2.5.3	对比线性模型和决策树模型的表现	77
2.5.4	回顾	79
2.6	模型融合	80
2.6.1	融合成群体 (Ensamble)	80
2.6.2	Bagging: 随机森林 (Random Forest)	82



2.6.3	Boosting: GBDT .....	83
2.6.4	Stacking .....	86
2.6.5	泰坦尼克号生存预测: 小结 .....	93
2.6.6	回顾 .....	94
<b>第3章 实战: 股票量化 .....</b>		<b>95</b>
3.1	第一步: 构造童话世界 .....	95
3.1.1	股票是什么 .....	95
3.1.2	当机器学习与量化交易走在一起 .....	96
3.1.3	构造一个童话世界 .....	96
3.1.4	回顾 .....	100
3.2	第二步: 应用机器学习 .....	100
3.2.1	构建特征数据 .....	100
3.2.2	回归预测股票价格 .....	103
3.2.3	分类预测股票涨跌 .....	108
3.2.4	通过决策树分类, 绘制决策图 .....	112
3.2.5	回顾 .....	114
3.3	第三步: 在真实世界应用机器学习 .....	114
3.3.1	回测 .....	115
3.3.2	基于特征的交易预测 .....	119
3.3.3	破灭的童话——真实世界的机器学习 .....	122

## 第二篇 深度学习篇

<b>第4章 深度学习: 背景和工具 .....</b>		<b>126</b>
4.1	背景 .....	126
4.1.1	人工智能——为机器赋予人的智能 .....	126
4.1.2	图灵测试 .....	126
4.1.3	强人工智能 vs 弱人工智能 .....	127
4.1.4	机器学习和深度学习 .....	128
4.1.5	过度的幻想 .....	128
4.1.6	回顾 .....	129

4.2	深度学习框架简介 .....	129
4.2.1	评测方式 .....	130
4.2.2	评测对象 .....	131
4.2.3	深度学习框架评测 .....	131
4.2.4	小结 .....	135
4.3	深度学习框架快速上手 .....	135
4.3.1	符号主义 .....	135
4.3.2	MNIST .....	136
4.3.3	Keras 完成逻辑分类 .....	138
4.3.4	回顾 .....	141
4.4	Caffe 实现逻辑分类模型 .....	141
4.4.1	Caffe 训练 MNIST 概览 .....	142
4.4.2	Caffe 简介 .....	144
4.4.3	准备数据集 .....	145
4.4.4	准备模型 .....	146
4.4.5	模型训练流程 .....	149
4.4.6	使用模型 .....	149
4.4.7	Caffe 的 Python 接口 .....	150
4.4.8	回顾 .....	151
第 5 章	深度学习模型 .....	152
5.1	解密生物智能 .....	154
5.1.1	实验一：大脑的材料 .....	154
5.1.2	实验二：探索脑皮层的功能区域 .....	156
5.1.3	实验三：不同的皮层组织——区别在于函数算法 .....	158
5.1.4	实验四：可替换的皮层模块——神经元组成的学习模型 .....	161
5.1.5	模拟神经元 .....	162
5.1.6	生物结构带来的启发 .....	163
5.1.7	回顾 .....	164
5.2	DNN 神经网络模型 .....	164
5.2.1	线性内核和非线性激活 .....	164
5.2.2	DNN、CNN、RNN .....	165
5.2.3	逻辑分类：一层神经网络 .....	166

5.2.4	更多的神经元 .....	167
5.2.5	增加 Hidden Layer (隐层) .....	168
5.2.6	ReLU 激活函数 .....	170
5.2.7	理解隐层 .....	171
5.2.8	回顾 .....	172
5.3	神经元的深层网络结构 .....	172
5.3.1	问题: 更宽 or 更深 .....	172
5.3.2	链式法则: 深层模型训练更快 .....	173
5.3.3	生物: 深层模型匹配生物的层级识别模式 .....	175
5.3.4	深层网络结构 .....	177
5.3.5	回顾 .....	178
5.4	典型的 DNN 深层网络模型: MLP .....	178
5.4.1	优化梯度下降 .....	179
5.4.2	处理过拟合: Dropout .....	181
5.4.3	MLP 模型 .....	182
5.4.4	回顾 .....	185
5.5	Caffe 实现 MLP .....	185
5.5.1	搭建 MLP .....	185
5.5.2	训练模型 .....	189
5.5.3	回顾 .....	190
第 6 章	学习空间特征 .....	191
6.1	预处理空间数据 .....	192
6.1.1	像素排列展开的特征向量带来的问题 .....	192
6.1.2	过滤冗余 .....	194
6.1.3	生成数据 .....	195
6.1.4	回顾 .....	198
6.2	描述图片的空间特征: 特征图 .....	199
6.2.1	图片的卷积运算 .....	199
6.2.2	卷积指令和特征图 .....	201
6.2.3	回顾 .....	206
6.3	CNN 模型 I: 卷积神经网络原理 .....	206
6.3.1	卷积神经元 .....	207

6.3.2	卷积层 .....	208
6.3.3	多层卷积 .....	211
6.3.4	回顾 .....	216
6.4	CNN 模型 II: 图片识别 .....	216
6.4.1	连接分类模型 .....	216
6.4.2	猫狗分类 .....	217
6.4.3	反思 CNN 与 DNN 的结合: 融合训练 .....	221
6.4.4	深度学习与生物视觉 .....	222
6.4.5	回顾 .....	224
6.5	CNN 的实现模型 .....	224
6.5.1	ImageNet 简介 .....	224
6.5.2	Googlenet 模型和 Inception 结构 .....	226
6.5.3	VGG 模型 .....	228
6.5.4	其他模型 .....	231
6.5.5	回顾 .....	232
6.6	微训练模型 (fine-tuning) .....	232
6.6.1	二次训练一个成熟的模型 .....	232
6.6.2	微训练在 ImageNet 训练好的模型 .....	233
6.6.3	回顾 .....	239
第 7 章	Caffe 实例: 狗狗品种辨别 .....	240
7.1	准备图片数据 .....	240
7.1.1	搜集狗狗图片 .....	240
7.1.2	清洗数据 .....	241
7.1.3	标准化数据 .....	242
7.1.4	回顾 .....	243
7.2	训练模型 .....	243
7.2.1	生成样本集 .....	244
7.2.2	生成训练、测试数据集 .....	245
7.2.3	生成 Imdb .....	246
7.2.4	生成去均值文件 .....	247
7.2.5	更改 prototxt 文件 .....	247
7.2.6	训练模型 .....	249

7.2.7 回顾 .....	249
7.3 使用生成的模型进行分类 .....	249
7.3.1 更改 deploy.prototxt .....	249
7.3.2 加载模型 .....	250
7.3.3 回顾 .....	257
<b>第 8 章 漫谈时间序列模型 .....</b>	<b>258</b>
8.1 Embedding .....	259
8.1.1 简单的文本识别 .....	260
8.1.2 深度学习从读懂词义开始 .....	261
8.1.3 游戏：词义运算 .....	264
8.1.4 回顾 .....	264
8.2 输出序列的模型 .....	265
8.2.1 RNN .....	265
8.2.2 LSTM .....	266
8.2.3 并用人工特征和深度学习特征——一个 NLP 模型的优化历程 .....	268
8.2.4 反思：让模型拥有不同的能力 .....	270
8.2.5 回顾 .....	273
8.3 深度学习：原理篇总结 .....	273
8.3.1 原理小结 .....	273
8.3.2 使用建议 .....	275
<b>第 9 章 用深度学习做个艺术画家——模仿实现 PRISMA .....</b>	<b>277</b>
9.1 机器学习初探艺术作画 .....	278
9.1.1 艺术作画概念基础 .....	278
9.1.2 直观感受一下机器艺术家 .....	279
9.1.3 一个有意思的实验 .....	280
9.1.4 机器艺术作画的愿景 .....	281
9.1.5 回顾 .....	282
9.2 实现秒级艺术作画 .....	282
9.2.1 主要实现思路分解讲解 .....	283
9.2.2 使用统计参数期望与标准差寻找 mask .....	290

9.2.3 工程代码封装结构及使用示例 .....	299
9.2.4 回顾和后记 .....	302
附录 A 机器学习环境部署 .....	303
附录 B 深度学习环境部署 .....	307
附录 C 随书代码运行环境部署 .....	312

# 第一篇 机器学习篇

机器学习篇主要面向机器学习零基础的读者，已有相关知识的读者可以直接跳过这一篇。

# 初识机器学习

本章将介绍几个浅层的学习模型，并尝试解释这些模型背后的“直观”原理。通过对本章的学习，相信你将有能力将这些模型运用到自己的工程中。

## 1.1 机器学习——赋予机器“学习”的灵魂

当人类用感情和希望去创造一样东西，那一样东西就会被赋予灵魂。

——宫崎骏《猫的报恩》

本节将对比机器学习和人类学习的过程。

400 多年前，我们发明了望远镜，拓展了“视觉”的能力；320 年前，我们发明了代步工具——自行车，提升了“步行”的能力；100 多年前，从热气球到莱特兄弟的飞机，我们具备了新的能力——“飞行”。在信息技术日益成熟的今天，机器学习将带我们步入更加神奇的世界——扩展“学习”的能力。

那么如何让机器像智能生物一样，获得学习的能力？早些时候的科学家一直试图让机械拥有真正意义上的智能，进而产生智能行为——学习，但最终这个方向并没有走通。今天的科学家走出了思维的桎梏，放弃纯粹的生物模仿，而是利用科学理论完成仿生——用数学模拟智能生物学习的过程。莱特兄弟发明的飞机是空气动力学的产物，而今天的机器学习是以数据为中心，通过训练模型发现数据中的某些内在模式，并将其运用到新数据中的技术。简单来说，机器学习就是研究数据，模拟生物学习的能力。

让我们先看一个简单的故事，直观对比一下机器学习和人类学习过程中的差异。

### 1.1.1 小红帽识别毒蘑菇

小红帽去森林采蘑菇，她希望自己能够了解哪些蘑菇是有毒的。第一天，采集了 5 朵蘑菇，小红帽观察蘑菇外表，发现其中 2 朵外表鲜艳的是毒蘑菇，3 朵外表朴素的是正



常蘑菇；于是小红帽学到一个新知识：“外表鲜艳的蘑菇是有毒的！”如图 1-1 所示。



图 1-1 毒蘑菇

小红帽通过眼睛观察蘑菇，机器则通过输入的数值识别事物。让机器模拟小红帽的这段学习经验，从用数据描述样本的信息开始：小红帽采集的 5 朵蘑菇，按特征（feature）提取数值，可以描述为

外表是否鲜艳

$$X : \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

注意，这里的数据行是不同的样本，数据列是“外表是否鲜艳”这一特征。其中编码信息：1-鲜艳，0-不鲜艳。得到的是否毒蘑菇的结果为

$$y : \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

其中，编码蘑菇的类别：1-毒蘑菇，0-正常蘑菇。 $y$  就叫作  $X$  的标签（label）向量，而 5 朵蘑菇样本数据  $X$  叫作训练集（training set）。机器从训练集学习到识别类别的知识，这一过程叫作模型（model）的训练，即：

机器学到的知识 == 训练好的模型

第二天，小红帽为了检验她的新知识，又采集了 5 朵新蘑菇做实验，结果发现误判了 2 朵蘑菇类别，于是小红帽得出结论：之前总结的知识识别蘑菇的准确率只有 60%。

对于学习到的模型，很自然地我们希望考察它在未知样本数据上面的应用能力，所

以接下来拿一部分和训练集不同的新样本，让模型预测。新的检验知识的蘑菇集合叫作测试集。

外表是否鲜艳

(test-set)  $X'$ :  $\begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix}$ , 对应的测试结果  $y'$ :  $\begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$ 。

小红帽通过“准确率”衡量模型的表现：

$$\text{准确率} = \frac{\text{正确分类的数量}}{\text{总数}}$$

小红帽反思了自己，仅仅通过“外表是否鲜艳”这一特征来辨别毒蘑菇是不可靠的。于是，小红帽学着从更多角度判读蘑菇是否有毒。例如，蘑菇的外表、生长地、尺寸。有的特征对辨识毒蘑菇很有帮助，有的则用处不大。慢慢地，小红帽学会了同时权衡这些特征，辨识一朵蘑菇是否有毒。毒蘑菇的生长环境如图 1-2 所示。



图 1-2 毒蘑菇的生长环境

从机器学习的角度来看：单个特征的模型太过简单，于是，提取了更多的蘑菇特征。

外表是否黏滑 生长地是否潮湿 大小尺寸 ...

$X = \begin{bmatrix} 0 & 1 & 3.3 & \dots \\ 1 & 1 & 7.3 & \dots \\ 0 & 0 & 1.3 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$ 。

接着，我们需要某种方式将这些特征组合起来，让它们一起发挥作用，帮助小红帽识别毒蘑菇。机器学习中的模型组合特征的方式有两类：

- 非参数化（如 1.2 节将要登场的 KNN）
- 参数化

非参数化的模型不多，1.2 节会简单介绍下，本书的重点是解释参数化的模型。采用参数化的方式组合特征就是：对于每个特征向量，通过模型的训练获得一个对应的参数矩阵（或者向量） $w$ ，模型的输出标签表示为： $y = \text{Model}(w, x)$ 。

在每个模型中， $w$  和  $x$  的组合方式都不一样。最简单的例子，可以对每个特征设置一个权重，权重越大，表示这种特征对目标类别影响越大。

$$\text{即 } w = \begin{bmatrix} \text{外表是否黏滑} & 0.7 \\ \text{生长地是否潮湿} & 0.2 \\ \text{大小尺寸} & -0.3 \\ \vdots & \vdots \end{bmatrix}$$

对应的类别输出： $y = x \cdot w$ ，( $x: M \times N, w: N \times 1$ )。总之，参数化的模型通过某种数学方式，权衡特征，输出判断结果。

那么机器究竟如何设置每个特征的权重呢？这个过程可以想象一下对狗的训练过程：当狗做对一件事情，我们就给它一根骨头，反之做错，略微惩罚它一下（设计这个惩罚方式的就叫作损失函数，后续我们会解释）。机器学习训练模型的过程也与之类似：将辨识样本是否正确作为修正信号。机器正确辨识时，对应的起作用的特征权重加分，错误辨识时对应的作用特征权重减分。最终，通过训练找到合适的权重，并按一定方式组合起来，得到一个可靠的分类器模型。

总结一下，小红帽学习的过程有三步：学习知识、修正知识、应用。机器学习的流程与之完全相同，如图 1-3 所示。

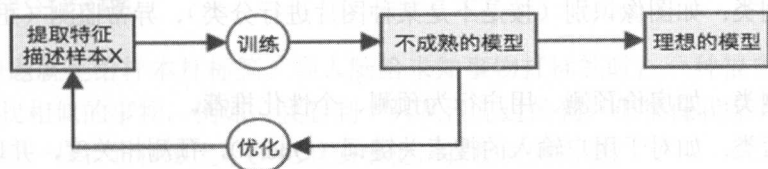


图 1-3 机器学习流程

- 训练模型
- 优化模型
- 应用模型

可以看到：从设计的角度来看，机器学习的实现并不是一种完全创新的“高深”思路，而是用数学的方式实现对自然智能行为的模拟。所有的机器学习算法，虽然数学推导和里层的代码实现很复杂，但背后往往都有非常简单的自然原型。对于应用层的工程师来说，理解其原型，运用其 API 就可以了（往往只有几行代码）——希望大家在阅读本书时，

带着这个观点。

### 1.1.2 三种机器学习问题

机器学习按应用的场景可以分为三类：

- 有监督机器学习 (Supervised Learning)
- 无监督机器学习 (Unsupervised Learning)
- 强化学习 (Reinforcement Learning)

给定数据集和对应的标签： $X - y$ ，训练模型，预测输出，这是有监督机器学习。本质上，机器学习模型只是在机械地拟合数据内在关系的表达式，赋予模型应用层面意义的是  $y$  值的含义。给定一个蘑菇的颜色、尺寸等信息，当我们设置  $y$  值代表“是毒蘑菇还是普通蘑菇”这一类别时，这就是一个分类任务 (Classification)；而当  $y$  值设置为“蘑菇的重量”时，这就变成了一个“预测蘑菇的重”的回归任务 (Regression)。

不关心有没有标签  $y$ ，只是挖掘数据集  $X$  的一些内在规律，这是无监督机器学习。本书不覆盖这一类问题。

在一些固定的场景下，机器在环境 (Environment) 中学习策略 (Strategy)，按策略选择一个动作 (Action)，目标是让对应的回报 (Reward) 最大，这是强化学习研究的范畴。本书也不覆盖这一类问题。

很多现实中的问题都可以转化成分类/回归问题，比如：

- 识别类：如图像识别（按是不是某种图片进行分类）、异常监测（正常类、异常类）。
- 预测类：如房价预测、用户行为预测、个性化推荐。
- 搜索类：如对于用户输入的搜索关键词 (Query)，预测相关度，并以排序后的结果返回。

接下来将专注于有监督机器学习问题，你将进一步看到这些模型算法是如何巧妙地实现对人类学习过程的模拟。

### 1.1.3 常用符号

本书常用的符号的含义如下：

- $M/m$ ：样本数量。
- $N/n$ ：一般指特征数量。

- $X$ : 样本集矩阵, 行样本标号列特征。
- $x$ : 一般指单个样本的特征值向量, 一些情况下也会仅仅表示输入 (Input)。
- $y$ : 输出值, 如果是模型最终的输出, 一般会 是样本的预测数值或者向量; 有时也会仅仅表示输出 (Output)。

### 1.1.4 回顾

- 机器学习算法是用数学模拟人类学习的过程。
- 单个特征是不可靠的, 但可以通过某种参数化的方式组合在一起, 形成可靠的经验知识——模型。
- 机器学习算法背后的直观解释, 往往非常清晰简单。

## 1.2 KNN——相似的邻居请投票

物以类聚, 人以群分。

——《易经》

KNN 模型 (K-Nearest Neighbor, K-近邻算法) 是一个典型的非参数模型, 也就是说, 计算机通过 KNN 学到的知识并不是以数值权重的方式存储下来的。本节将通过 KNN 模型, 进一步熟悉机器学习技术。

### 1.2.1 模型原理

分类问题就是给样本打标签。当人脑给未知事物打标签时, 一种很自然的做法就是在脑海中寻找相似的事物。例如, 我看到一只从没见过的狗, 但很像我家养的拉布拉多犬, 那么我说这是一只拉布拉多犬——相似的事物, 有相同的标签。

KNN (K-近邻算法) 便是这一思想的一种实现, 可用于分类问题和回归问题, 简单而有效。在分类问题上, KNN 就是选出与待分类样本最相似的  $K$  个近邻, 投票确定样本所属的类别。近邻分类如图 1-4 所示。

KNN 既然是  $K$  近邻投票选类别, 那么这里便引出两个问题:

- 如何选出  $K$  近邻。
- 如何“投票”, 或者说, 选票的权重是否相等。

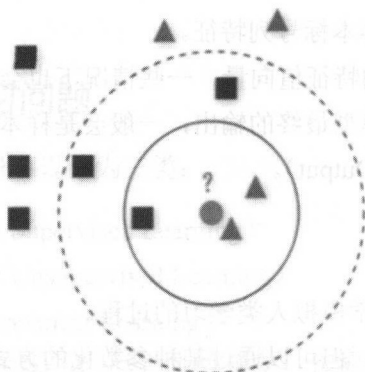


图 1-4 近邻分类 (出自维基百科)

### 1. 如何选出 K 近邻

之所以称呼为“近”邻，是因为我们用欧氏距离衡量样本之间的相似程度，距离越大越不相似。样本点  $a$ 、 $b$  的距离公式：

$$(a, b) = \sqrt{\sum_{i=0}^n (a_i - b_i)^2}$$

( $i$ : 样本标号)

代码实现：

```
import numpy as np # NumPy 可以快速操作结构数组

vec1 = np.array([1, 2, 3])
vec2 = np.array([4, 5, 6])
# 欧氏距离
assert np.linalg.norm((vec1 - vec2)) == np.sqrt(
    np.sum(np.square(vec1 - vec2)))
```

在 KNN 中，具体执行步骤如下：

- (1) 计算待测样本和训练集中每个样本点的欧式距离。
- (2) 对上面所有的距离值排序。
- (3) 选前  $k$  个最小距离的样本作为“选民”。

这些步骤在代码实现层中都封装在了 API 函数内部，并没有暴露给上层的使用者，所以大家知道有这么一回事就可以了。

## 2. 投票

KNN 算法中，所选择的邻居都是已经正确分类的对象。基本上邻居们是均匀权重投票，就是我们常规意义理解的一人一票，计数比较结果。但如果在特定场景下，也可以设置权重规则。scikit-learn 中通过设置 `weights` 参数选择对应的权重算法。

下面看一个具体的例子。

### 1.2.2 鸢尾花卉数据集 (IRIS)

埃德加·安德森在加拿大加斯帕半岛上调研提取了鸢尾属花朵的地理变异数据。数据集  $X$  包含了 150 个样本 ( $M = 150$ )，都属于鸢尾属下的三个亚属 (3 个分类类别)，分别是山鸢尾 (*setosa*)、变色鸢尾 (*versicolor*) 和维吉尼亚鸢尾 (*virginica*)，如图 1-5 所示。数据集通过四个特征维度 ( $N = 4$ ) 描述鸢尾属花朵样本：花萼长度 (`sepal_length`)、花萼宽度 (`sepal_width`)、花瓣长度 (`petal_length`)、花瓣宽度 (`petal_width`)。

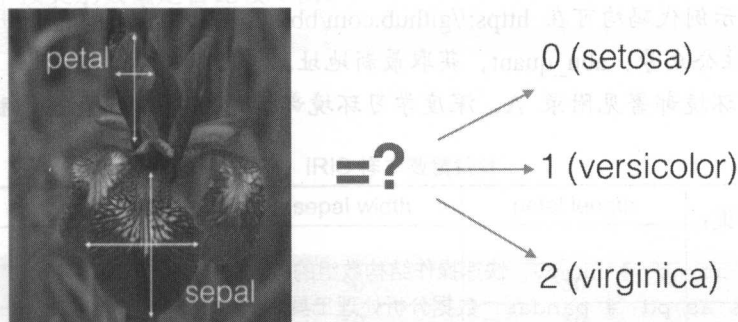


图 1-5 IRIS 样本分类

我们将通过这些数据学习到一个可用于鸢尾花卉属性类别的分类器。

### 1.2.3 训练模型

#### 第一步：描述任务

问题：

- 150 个样本。
- 四个特征维度。
- 花萼长度 `sepal length (cm)`。
- 花萼宽度 `sepal width (cm)`。
- 花瓣长度 `petal length (cm)`。



- 花瓣宽度 petal width (cm)。
- 目标：3-Class 样本 (0-setosa, 1-versicolor, 2-virginica)。

目标：

获得 KNN 分类器，在给定样本特征的未知标签数据上（如  $x = [3 \ 2 \ 5 \ 3]$ ），可以准确预测标签  $y$ 。

第二步：观察数据集

我们重点观察了两个方向：

- 采集的样本是否有缺失数据。
- 不同类别的样本数量分布是否基本均匀。

### 说明

本书所有示例代码均可在 <https://github.com/bbfamily/abu> 中下载并使用，如下载地址有变动，请关注公众号：abu\_quant，获取最新地址。

机器学习环境部署见附录 A，深度学习环境部署见附录 B，随书示例代码运行环境部署见附录 C。

加载数据集：

```
import numpy as np # numpy: 快速操作结构数组的工具
import pandas as pd # pandas: 数据分析处理工具
import matplotlib.pyplot as plt # matplotlib: 画图工具
from sklearn import datasets # datasets: sklearn 的示例数据集

# 数据集 0-setosa、1-versicolor、2-virginica
scikit_iris = datasets.load_iris()
# 转换成 pandas 的 DataFrame 数据格式，方便观察数据
iris = pd.DataFrame(
    data=np.c_[scikit_iris['data'], scikit_iris['target']],
    columns=np.append(scikit_iris.feature_names, ['y']))
```

观察数据格式：

```
iris.head(2)
```

输出如表 1-1 所示。



表 1-1 IRIS 数据格式

sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	y
5.1	3.5	1.4	0.2	0.0
4.9	3.0	1.4	0.2	0.0

检查数据是否有缺失：

```
iris.isnull().sum()
```

输出：

```
sepal length (cm)    0
sepal width (cm)     0
petal length (cm)    0
petal width (cm)     0
y                    0
dtype: int64
```

观察样本中按类别数量是否比较均衡：

```
iris.groupby('y').count()
```

输出如表 1-2 所示。

表 1-2 IRIS 样本数量统计

	sepal length	sepal width	petal length	petal width
y				
0.0	50	50	50	50
1.0	50	50	50	50
2.0	50	50	50	50

可以看到，IRIS 数据样本很理想、类别分布均匀，基本不需要任何预处理。

### 第三步：训练模型

scikit-learn（简称 sklearn）是 Python 最为流行的一个机器学习库。它的特点：

- 开源。
- 上手简单、文档丰富、性能高效、算法全面。
- 生态系统完整，基于 NumPy、SciPy。以及 Matplotlib，从数据探索性分析、数据可视化到算法实现，整个过程一体化实现。

在本例中，应用 scikit-learn 的 KNN 实现只需 3 步：选择模型、训练数据、预测。



```
# K=15
knn = KNeighborsClassifier(n_neighbors=15)
knn.fit(X_train, y_train)

y_pred_on_train = knn.predict(X_train)
y_pred_on_test = knn.predict(X_test)
# print metrics.accuracy_score(y_train, y_pred_on_train)
print 'accuracy: : {}'.format(
    metrics.accuracy_score(y_test, y_pred_on_test))
```

输出:

```
accuracy: : 0.973684210526
```

## 2. abupy

这里介绍一下笔者开源的 Python 库——abupy。abu 量化系统包括数据、alpha、beta、因子、交易以及 MLBu 机器学习等模块。其中 MLBu 机器学习模块对于一些常用模型的训练、测试以及评估的实现代码进行了二次封装，节省了一些重复工作。

abupy 中集成了 IRIS 数据集，以及一些其他示例数据集。读者可以在 <https://github.com/bbfamily/abu> 中下载并使用 abupy，如下载地址有变动，可通过关注公众号:abu\_quant 获取最新地址，abupy 安装教程详见附录 A。

输入:

```
from abupy import AbuML

# IRIS 花卉数据集
iris = AbuML.create_test_fiter()

# 使用 KNN
iris.estimator.knn_classifier(n_neighbors=15)

# cross-validation 测试
iris.cross_val_accuracy_score()
```

输出:

```
accuracy mean: 0.973333333333
array([ 0.93333333,  0.93333333,  1.          ,  1.          ,  1.          ,
        0.93333333,  0.93333333,  1.          ,  1.          ,  1.          ])
```

上面的类 AbuML 是对机器学习模型的通用封装，cross\_val\_accuracy 是比分割训练-测试集更为复杂的测试方式，详见 2.2 节。下面展开中间库的一部分相关的代码供大家参考：

```

class AbuML(object):
def __init__(self, x, y, df, force_clf=False):
    """
        中间层需要所有原料都配齐
    """
    self.estimator = AbuMLCreator()

    self.x = x
    self.y = y
    self.df = df
    self.force_clf = force_clf

@classmethod
def create_test_fiter(cls):
    """
        IRIS 花卉数据集
    """
    iris = load_iris()
    x = iris.data
    y = iris.target

    x_df = pd.DataFrame(x, columns=['x0', 'x1', 'x2', 'x3'])
    y_df = pd.DataFrame(y, columns=['y'])
    df = y_df.join(x_df)
    return AbuML(x, y, df)

# .....省略部分函数

def _do_cross_val_score(self, x, y, cv, scoring):
    fiter = self.get_fiter()

    scores = cross_validation.cross_val_score(fiter, x, y, cv=cv,
                                              scoring=scoring)

    mean_sc = np.mean(
        np.sqrt(-scores)) if scoring == 'mean_squared_error' \
        else np.mean(scores)

    logging.info(scoring + ' mean: ' + str(mean_sc))
    return scores

```

## 1.2.5 关于 KNN

KNN 是一种简单而且有效的算法，但是 KNN 在对测试集进行分类时需要训练集样本（算测试样本的  $K$  近邻），所以使用算法时必须有接近实际数据的训练样本数据，而且必须保存全部数据集。如果训练数据集很大，则必须使用大量的存储空间。此外，由于必须对数据集中的每个数据计算距离值，因此实际使用时在模型预测这一步可能非常耗时——这一点是限制 KNN 推广的最大弊端。



KNN 特点:

- 原理简单。
- 保存模型需要保存全部样本集。
- 训练过程很快, 预测速度很慢。

后续将介绍一些基于参数表示的分类模型, 这些模型的保存不需要样本集, 只是存储一些参数矩阵和权值。

## 1.2.6 运用 KNN 模型

本小节主要面向有经验的读者, 谈谈模型理解及实际运用, 新入门的朋友可以暂时忽略这里, 有一定实践基础后再来阅读。

《塔希里亚故事集》中有一句经典的语录: “世上没有低级的法术, 只有低级的法师”。KNN 虽然是个非常简单的模型, 但在很多数据任务中如果被有效使用, 也可以拿到惊人的效果。我的朋友就在一个 NLP 文本内容分类的国际级比赛中, 使用 KNN 模型获得了第二名。

在运用 KNN 模型时请注意以下几点。

### 1. 发扬 KNN 模型的特性

近邻分类的思路使 KNN 在近邻聚集比较好的数据任务中表现良好, 也就是说, 如果一个数据任务类别清晰。如一段文本是新闻还是娱乐, 一般是很清晰的, 可以考虑运用 KNN; 反之, 如果任务类别不清晰, 如预估用户是否点击一个事件, 这是很随机的, 或者在股票等混沌市场做一些预测, 这种任务 KNN 表现一般不会太好。

### 2. 规避工程执行的短处

KNN 的工程问题在于预测时时间复杂度高, 本书后面登场的参数化模型就没有这个问题。工程上在大数据量上使用 KNN 时, 需要解决这一问题。对于稀疏特征值的数据, 如词文本, 可以考虑做索引表, 记录词的出现位置, 拿存储代价换计算代价; 密集特征值数据可以考虑 kd-tree 等数据结构。

### 3. KNN 的突破点

从 KNN 的近邻投票可以看出, 优化模型时, 突破口在于相似度的计算, 就是如何合理地划分出近邻。根据数据任务特性, 创意地设计相似度计算方式, 可以使 KNN 模型发挥出惊人的能量。朋友的 NLP 比赛当时就是创意地在相似度计算中融入了单词的熵值,

获得了不错的效果。

所以，对于每个机器学习模型，运用时请注意以下三点：

- 思考并发扬模型的特性。
- 规避工程执行的短处。
- 优化时寻找模型的关键位置。

## 1.2.7 回顾

祝贺你成功迈出机器学习的第一步！

回顾一下看看机器学习究竟做了什么？我们采集了一些花卉特征数据（训练集）——这是程序的输入；希望可以训练出某种模型，在新的花卉数据（测试集）上自动识别出它们的类别（标签）——这是目标输出；同时，我们还想到可以用相似的事物标签相同这一思想实现分类——这是任务的实现思路。有了输入、目标输出以及实现思路，接着搭建出对应的执行程序。从程序的角度来看，所谓模型，就是将输入的特征数据组合起来，输出期望的结果的一套自动化程序。不同模型的区别就在于组合输入数据的方式不同。比如，KNN 模型采取了一种直观思路（物以类聚）的方式组合特征输入，后续的其他模型则是带着同样的使命，通过其他的奇思妙想去组合特征，输出结果。

Follow us，未来还有更多机智而且有趣的模型等待着你！

## 1.3 逻辑分类 I：线性分类模型

给我五个系数，我将画出一头大象；给我六个系数，大象将会摇动尾巴。

——AL 柯西

本节介绍逻辑分类模型，本书只解释模型的直观原理，模型实现的代码一般只有几行，非常简单。

### 1.3.1 参数化的模型

正如 1.2 节所提到的观点：模型将输入特征数值组合起来，输出期望的结果。我们可以借鉴一些自然直观的分类思路组合输入的数值，如 KNN 的相似者同类，也可以用数学表达式的方式组合输入数值。

举个简单的例子——识别毒蘑菇。特征有两个：是否鲜艳  $x_1$ ，是否生长在阴暗的

地方  $x_2$ 。我们用线性数学表达式的方式组合，比如我觉得“蘑菇是否鲜艳”这一特征对判别毒蘑菇分类的作用大约是“蘑菇是否生长在阴暗地”的 3 倍。那么我们可以按  $\text{score} = 3x_1 + x_2 - 0.4$  计算一个样本的分数。其中  $-0.4$  可以认为是基础分，因为我们觉得毒蘑菇比正常少一点，所以“一般情况”下认为不是毒蘑菇。计算得出的  $\text{score}$  值越大，我们猜测这是朵毒蘑菇的可能性越大，大到一定程度，我们就认定这一定是朵毒蘑菇。

在这里， $-0.4$ 、 $x_1$  和  $x_2$  的乘法系数

	是否鲜艳	3.0
$w = [$	生长地是否阴暗	1.0]
	一般情况下是否有毒	-0.4

叫作特征的权重；这些权重代表模型认为特征对分类结果的影响程度。权重越大，说明这一特征对分类结果影响越大。用带权重的数学表达式的方式组合输入特征数值——这就是参数化的模型。对比 KNN，模型对事物认知的表达方式从依靠近邻的数据集迁移到了参数矩阵，如图 1-6 所示。

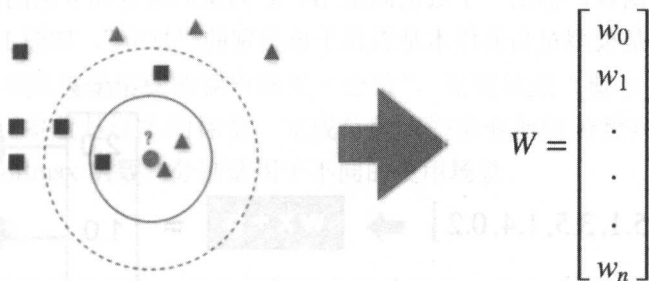


图 1-6 认知方式的迁移

参数化的模型还有其他一些好处——更低存储、更快的预测速度。因为模型对事物的识别方式变成了依赖数学表达式和对应的权重矩阵，所以，对于训练好的模型存储变得异常方便。回顾 1.2 节 KNN 在应用分类时需要训练集样本（算待测样本的  $K$  近邻），所以训练好的模型在保存时，要保存全部训练数据，并且单个样本测试的时间复杂度是  $O(n)$  级别的。而参数化模型的存储不再依赖训练集数据，直接保存参数矩阵就可以了，大大节省了空间；并且在预测数据样本时只需将样本数据代入数学表达式就可以了，单个样本测试的时间复杂度降到  $O(1)$ 。相比 KNN 的快训练慢预测，大多数场景下我们肯定更愿意接受训练学习时慢一点，在应用模型做服务时快速响应。

### 1.3.2 逻辑分类：预测

把每个特征对分类结果的“作用”加起来——这就是线性模型。逻辑分类（Logistic Classification）是一种线性模型，可以表示为  $y = w \cdot x + b$ ，其中  $w$  是训练得到的权重参数（Weight）； $x$  是样本特征数据； $b$  是偏置（Bias），与上面的“一般情况下是否有毒”这一特征作用类似，表示基础类别倾向。需要说明的是，有些资料中逻辑分类也叫作逻辑回归（Logistic Regression），但它本身是用作分类问题的。

逻辑分类模型预测一个样本分为三步：

- (1) 计算线性函数。
- (2) 从分数到概率的转换。
- (3) 从概率到标签的转换。

#### 第一步：线性函数

我们继续以 IRIS 花卉数据集为例讲解，逻辑分类的第一步：对于某个输入样本  $x$ ，让它通过一个线性函数，输出一个数值向量  $S$ 。 $S$  向量的长度和分类的类别数量一致，其中，向量的每个值是模型对当前样本是否属于该类别的“打分”，如图 1-7 所示。

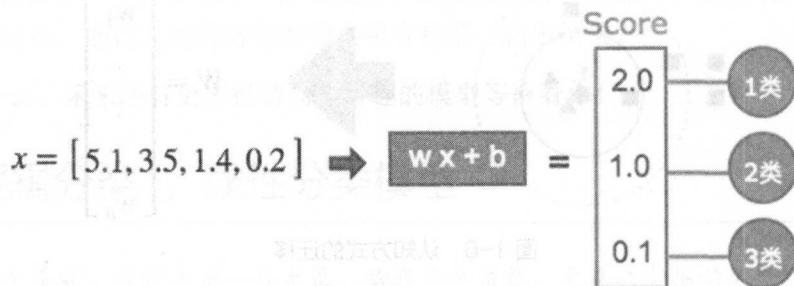


图 1-7 逻辑分类打分样本

“打分”函数：

```
import numpy as np

def score(x, w, b):
    return np.dot(x, w) + b
```

对于逻辑分类，训练模型的目标就是找到合适的  $w$  和  $b$ ，让输出的预测值变得可靠。举个简单的例子，下面将输入的特征数值打印在平面坐标系上，那么线性表达式的  $w$  和  $b$  就好比一条直线的斜率和截距，这两个参数会调整直线的位置，让直线变得尽可能接近所有的样本点，如图 1-8 所示。



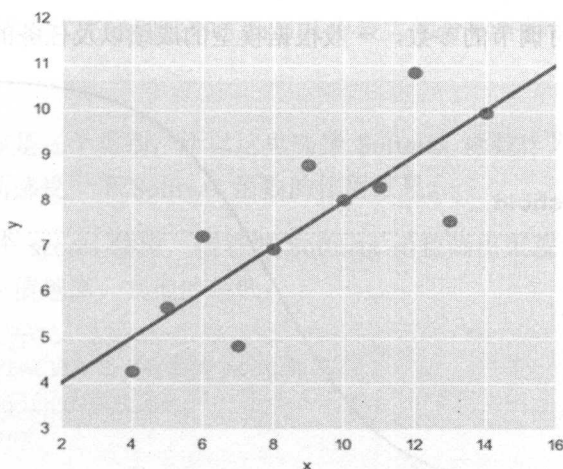


图 1-8 拟合直线

## 第二步：将分数变成概率

首先，我们将输入数据经过线性函数，得到一个分数向量，这个向量如何变换成标签化的结果呢？

答案是我们将线性表达式的输出结果“分数”，先变换成“概率”，再对应标签——概率最高的类别标签就是样本的标签。完成分数到概率变换的函数有很多种，常用的是 Sigmoid 函数和 Softmax 函数，分别适用于不同的使用场景。

### Sigmoid 函数

Sigmoid 函数适用于只对一种类别进行分类的场景。首先设置函数阈值（Shreshold），当 Sigmoid 函数输出值大于阈值，则认为“是”这一类别；否则认为“不是”这一类别。

$$\text{Sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

Sigmoid 函数：

```
def sigmoid(s):
    return 1. / (1 + np.exp(-s))
```

如图 1-9 所示。

直观地讲，Sigmoid 函数就做了两件事：

- 将输入的“分数”的范围映射在 (0, 1) 之间。
- 以“对数”的方式完成到 (0, 1) 的映射，凸显大的分数的作用，使其输出的概率更高；抑制小分数的输出概率。

函数阈值是一个可调节的参数，一般根据模型的成绩以及任务的需求调试。

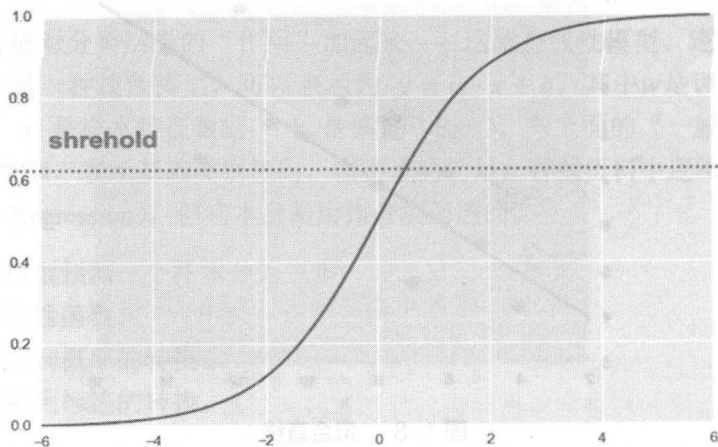


图 1-9 Sigmoid 函数

### Softmax 函数

Softmax 函数是 Sigmoid 函数的“多类别”版本，可以将输出值对应到多个类别标签，概率值最高的一项就是模型预测的标签。

$$\text{Softmax}(s) = \frac{e^{s_i}}{\sum_j e^{s_j}}$$

Softmax 函数：

```
def softmax(s):
    return np.exp(s) / np.sum(np.exp(s), axis=0)
```

如图 1-10 所示。

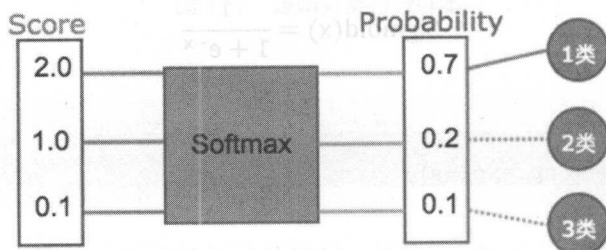


图 1-10 使用 Softmax 转换成概率

同样，Softmax 函数也做了两件事：

- 将输入的“分数”的范围映射在 (0, 1) 之间，并且所有分数的和为 1。

- 以“对数”的方式完成到映射，凸显其中最大的分数并抑制远低于最大分数的其他数值。

由于 IRIS 的标签是三个类别，所以这里选择 Softmax 函数作为分数—概率转化函数。我们可以通过代码侧面感受一下 Softmax 函数的作用效果。

(1) 随机模拟一个 scores 输出，接着把 Softmax 函数画出来观察：当  $x$  的数值越大时，第一个分类的概率值越高，其他的越低：

```
import matplotlib.pyplot as plt
# Seaborn是一个Matplotlib之上封装的plot类库
# 这里我们只是使用Seaborn的样式定义
import seaborn as sns

x = np.arange(-3.0, 6.0, 0.1)
scores = np.vstack([x, np.ones_like(x), 0.2*np.ones_like(x)])
plt.plot(x, softmax(scores).T, linewidth=2)
```

输出如图 1-11 所示。

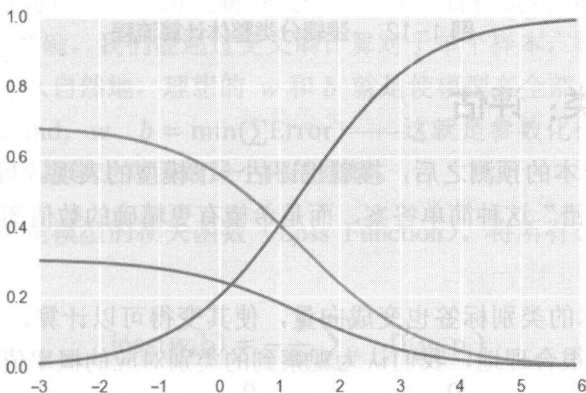


图 1-11 Softmax 函数

(2) 将分数扩大/缩小 100 倍，观察输出的概率值变化：

```
scores = np.array([2.0, 1.0, 0.1])
print softmax(scores)
print softmax(scores * 100)
print softmax(scores / 100)
```

输出：

```
[ 0.65900114  0.24243297  0.09856589]
[ 1.00000000e+00  3.72007598e-44  3.04823495e-83]
[ 0.33656104  0.33321221  0.33022675]
```

分数扩大 100 倍后，概率值大的越大，小的越小。即分类器对分类的结果更加“自信”；反之缩小 100 倍后，分类器显得对分类的结果很“犹豫”。

### 第三步：从概率到类别

到这里就很简单了，我们选择概率最高的类别作为预测的类别标签。

总结一下，逻辑分类分为三步完成预测样本分类的任务。

(1) 经过线性函数，将  $x$  变换成在不同类别上的预测“分数”。

(2) 通过 Softmax 函数（我们会在下文中有更详细的介绍），将每个类别上的分数转换成对应的概率。概率值越高，预测样本就越可能是这种类别。

(3) 将概率向量  $P$  变换成类别  $y$ ，选出概率值最高的那个维度就是模型预测的类别。

简而言之，逻辑分类的预测：线性函数→概率→类别，如图 1-12 所示。

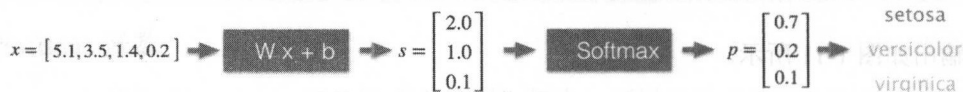


图 1-12 逻辑分类整体计算流程

### 1.3.3 逻辑分类：评估

当模型完成对样本的预测之后，接着想评估一下模型的表现。与 1.2 节不同，我们不再满足预测的“对或错”这种简单答案，而是希望有更精确的数值衡量出预测样本和真实样本的表现差距。

首先需要让样本的类别标签也变成向量，使其变得可以计算。逻辑分类本身是按概率分类标签的，那么很合理地，我们认为观察到的类别对应的概率值就是 1，其他是 0。

即：setosa =  $\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ 、versicolor =  $\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$ 、virginica =  $\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$

这种编码  $y$  值的方式叫作 One-Hot 编码。

接下来，对于一个样本  $x$ ，我们手上已有两个概率分布向量：一个是待测样本的分类概率向量： $P = \begin{bmatrix} 0.7 \\ 0.2 \\ 0.1 \end{bmatrix}$ ，来自模型，代表模型对样本的评估；另一个来自真实样本的标签：

$y = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ 。我们的目标是衡量这两个概率分布向量的差异程度。

在统计学中，衡量两个概率分布向量的差异程度，叫作交叉熵（Cross Entropy）。

说明：熵是信息的别称，关于信息见 2.5 决策树模型——信息与编码。

$$D(y, p) = y \ln(p) + (1 - y)(1 - \ln(p)) \quad (\text{其中 } p = \text{softmax}(\text{score}(wx + b)))$$

交叉熵函数：

```
# y 是真实标签, p 是预测概率
def cross_entropy(y, p):
    return np.sum(y * np.log(p) + (1 - y) * np.log(1 - p, axis=1))
```

和之前的欧氏距离的作用不同，交叉熵是衡量两种概率分布相同的“概率”，数值在 (0,1) 之间，交叉熵越高越相似，为 1 则完全相同。

于是，我们就有了评估模型对一个样本的预测是否准确的数值标准：Error =  $-D(y, p)$ ，这个数值越小，对这个样本的预测越准。

### 1.3.4 逻辑分类：训练

现在，我们可以讨论如何训练逻辑分类的模型了。目标是获得“合适”的  $w$  和  $b$ ，让模型的预测尽可能准确。我们能通过交叉熵计算对于单个样本，模型的预测值和真实值之间的差距 Error。那么自然地，理想的  $w$  和  $b$  就是使模型在全部训练数据上的预测差距之和尽可能的小。即 find:  $w, b = \min(\sum \text{Error})$ ——这就是参数化模型训练的核心内容，将模型的训练问题变成数学上对函数最小化的求解。

这里，交叉熵就是模型的损失函数（Loss Function）。将所有公式对接展开，对于训练集  $X$ ：

$$\text{loss}(w, b) = -\frac{1}{M} \sum_i D(y_i, p_i)$$

其中， $i$  是每个样本的标号， $p_i = \text{softmax}(\text{score}(wx_i + b))$ 。

损失函数：

```
# x 是训练样本矩阵, w 是权重向量, b 是偏置向量, y 是真实标签矩阵
def loss_func(X, w, b, y):
    s = score(X, w, b)
    y_p = softmax(s)
    return -np.mean(cross_entropy(y, y_p))
```

那么计算机如何找到理想的  $w, b$ ，从而使  $\text{loss}(w, b)$  尽可能小呢？这并不是一个难题，我们可以选择最原始的方式——尝试暴力搜索所有  $w$  和  $b$  的权重参数组合，选择使

$\text{loss}(w, b)$  数值最小的  $w$  和  $b$  的组合即可。实际上，因为数字的组合方式有无限多种，我们并不可能尝试所有组合，所以 1.4 节会介绍一些“小技巧”，从而让计算机更加聪明地寻找恰当的参数组合。

### 1.3.5 回顾

- 对比 KNN，参数化模型，慢训练快预测。
- 线性模型：把特征对分类结果的作用按权重比例加起来，用特征的权重矩阵表示模型对事物的认知。
- 逻辑分类的预测分为三步，线性函数→概率→分类。
- One-Hot 编码、交叉熵。
- Softmax 函数：将分数变成概率。
- 逻辑分类模型训练的目标就是求解使损失函数  $\text{loss}(w, b)$  最小的参数组合  $(w, b)$ 。

## 1.4 逻辑分类 II：线性分类模型

我知道结果，之所以在犹豫，只是在找寻路径。

——《疯狂动物城》

本节介绍一些让逻辑回归模型训练得更好的技巧。

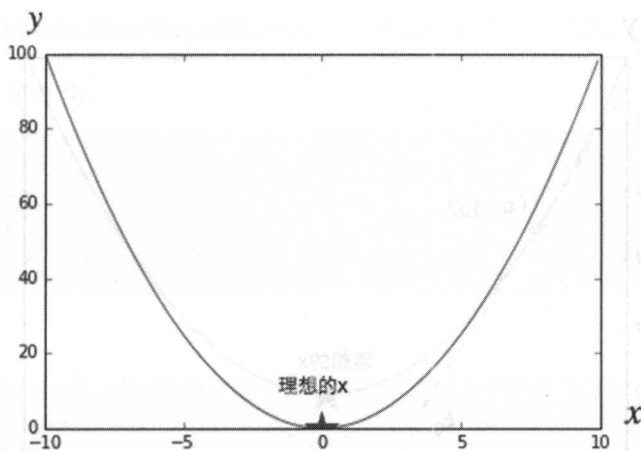
### 1.4.1 寻找模型的权重

1.3 节提到了逻辑分类在模型训练这一环节的目标就是要找到理想的  $w$ 、 $b$ ，使损失函数： $\text{loss}(w, b) = -\frac{1}{M} \sum_i D(\text{score}(wx_i + b), y_i)$  尽可能的小。要想达成这一目标，最简单的想法是用计算机暴力搜索所有可能的权重参数组合，但世界上有无限多的数字，即有无限多的数值组合，一个个尝试显然是不可能的。因此，我们需要设计一种实现思路，让计算机能够学会用更加聪明的方式去寻找理想的参数组合，而不是没头没脑地一个个尝试。

#### 灵感：偏导数

让我们先简化问题，寻找一些设计灵感。对于一个单一变量的函数  $y = f(x)$ ，函数图如图 1-13 所示，解决其最小问题。即寻找解  $x$ ，使函数  $y$  最小。



图 1-13 函数  $f(x)$ 

定义函数:

```
import numpy as np

# 函数  $y = 1 \cdot x^2 + 0 \cdot x + 0$ 
y = np.poly1d([1, 0, 0])
y(-7)
```

输出:

49

数学上, 解决这个问题的方法很简单——导数。对于图 1-13 中的函数, 求解上面的问题只需要求解  $\frac{\partial y}{\partial x} = 0$  即可。导数有如下一些性质:

- 函数趋势向上的区域, 导数大于 0。
- 函数趋势向下的区域, 导数小于 0。
- 在函数最小值处, 导数等于 0。

函数在一点的导数如图 1-14 所示。

测试导函数:

```
# d_yx: 导函数
d_yx = np.polyder(y)
d_yx(-7)
```

输出:

-14

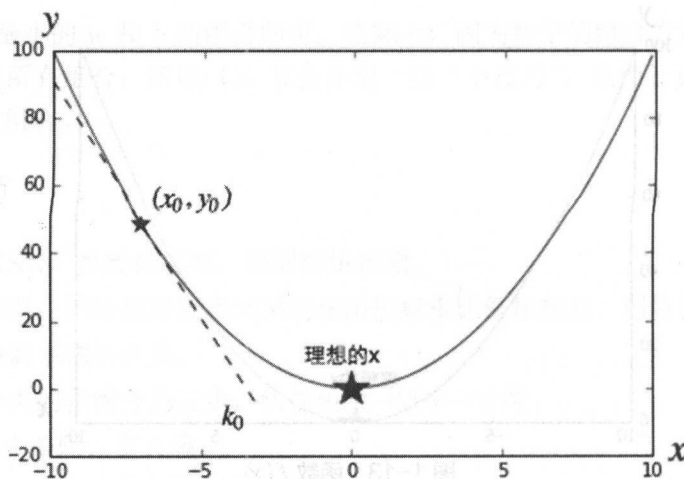


图 1-14 函数在一点的导数

这些性质给了我们灵感：对于寻找使函数最小的参数问题，导数有“引路”的作用。

然而现实中面对的真实问题远没有上面的函数那么理想，计算机面对的是离散的数据样本，而不是连续的数值，直接求解导数是困难的。回想本书开头提到的观点：机器学习做的事情就是用数学模拟智能行为——我们完全可以通过导数，设计一种实现流程，让机器智能地寻找使函数值最小的解。

(1) 随机选一个起点  $(x_0, y_0)$ ：

```
import random

# 随机选一个起点
x_0 = random.uniform(-10, 10)
y_0 = random.uniform(-10, 10)
x_0, y_0
```

输出：

```
(-4.292642173180106, -4.61100361485554)
```

(2) 计算当前点的导数，按导数“指引”，往前走一步，到达一个新点  $(x_1, y_1)$ 。  
即  $x_1 = x_0 - \alpha k_0$ ，这里  $k_0 = \frac{\partial y}{\partial x}$ ， $\alpha$  代表这一步的长度，叫作学习速率。

```
def step(x, d_yx):
    alpha = .2
    return x - alpha*d_yx(x)
step(x_0, d_yx)
```

输出：



```
-2.5755853039080634
```

(3) 不断重复第2步:

```
x = x_0
x_list = [x]
for i in range(10):
    x = step(x, d_yx)
    x_list.append(x)
x_list
```

输出:

```
[-4.292642173180106,
-2.5755853039080634,
-1.545351182344838,
-0.92721070940690276,
-0.55632642564414159,
-0.33379585538648493,
-0.20027751323189094,
-0.12016650793913455,
-0.072099904763480729,
-0.043259942858088436,
-0.02595596571485306]
```

可以看到,按上面设计的执行流程,随着 $x$ 越来越接近底部,导数的绝对数值越来越小,每一步 $x$ 的变化量也越来越小,最终寻找轨迹会在底部来回震荡,这个现象叫作收敛,如图1-15所示。

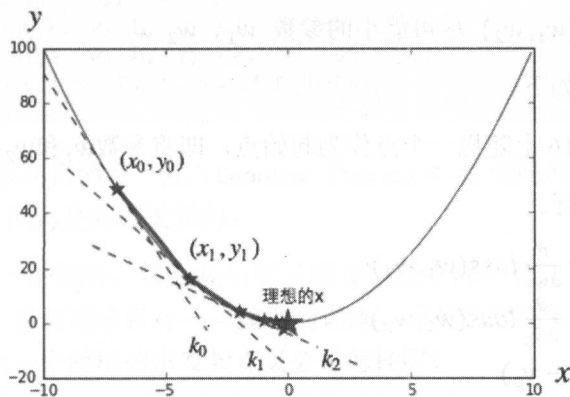


图1-15 收敛

就这样,计算机完成了对最佳参数 $x$ 的寻找过程。由于底部震荡,计算机给出的理想参数 $x'$ 不一定和最佳的参数 $x$ 完全相同,我们可以忽略这种损失,比起暴力尝试所有参数组合,能够在有限的执行步骤中找到一个“相对合适”的参数其实已经达成我们的目标了。

## 寻找模型的参数组合

现在我们有了底气，回到之前的模型训练问题，寻找  $w$ 、 $b$ ：使  $\min(\text{loss}(w, b))$ 。虽然函数的参数从之前的 1 个变成了 2 个参数向量，但我们依旧按之前的思路设计执行流程。为了更形象地说明这一方法，图 1-16 将参数  $w$  的两个维度抽出来： $w_1$ 、 $w_2$ ，画在了平面上。

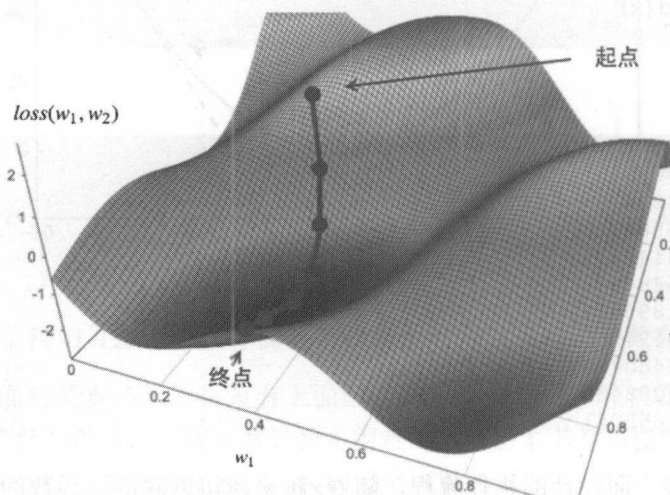


图 1-16 梯度下降寻找参数

计算机从一个随机起点开始，目标是寻找图中的“洼地”。计算机通过摸索出一条“路径”，找到使  $\text{loss}(w_1, w_2)$  尽可能小的参数  $w_1$ 、 $w_2$ 。

具体的执行过程如下。

第一步：在图 1-16 上随机一个点作为起始点，即将参数  $w_1$  和  $w_2$  初始为随机数。

第二步：迭代执行。

- $w_1 = w_1 - \alpha \frac{\partial}{\partial w_1} \text{loss}(w_1, w_2)$
- $w_2 = w_2 - \alpha \frac{\partial}{\partial w_2} \text{loss}(w_1, w_2)$
- $b = b - \alpha(y - y_p)$

直至  $\text{loss}(w_1, w_2)$  收敛。大家不用花太多时间关注公式细节，了解其大致原理就可以了，这些细节都封装在机器学习库的函数中，上层使用者并不需要掌握。

下面定义一个叫作“优化器 (Optimizer)”的类对象，帮助模型优化参数组合。

定义函数：

```
def d_loss_func(X, w, b, y, w_i):
    # 损失函数对 w 的偏导数
    s = score(X, w, b)
    y_p = softmax(s)
    return np.mean(w_i * (y_p - y))

def d_b_loss(X, w, b, y, d_obj=1):
    # 损失函数对 b 的偏导数
    s = score(X, w, b)
    y_p = softmax(s)
    return np.mean(d_obj * (y_p - y))

def step(X, w, b, y, d_obj, loss_func):
    # alpha 是一个可调节的模型参数
    alpha = .2
    return w_i - alpha * loss_func.__call__(X, w, b, y, d_obj)

class GDOptimizer:
    """梯度下降优化器"""
    def optimize(X, y):
        # 随机选一个起点
        w1 = random.uniform(0, 1)
        w2 = random.uniform(0, 1)
        b = random.uniform(0, 1)
        w = [w1, w2]
        # 迭代 100 次
        for i in range(100):
            w1 = step(X, w, b, y, w1, d_loss_func)
            w2 = step(X, w, b, y, w2, d_loss_func)
            b = step(X, w, b, y, b, d_b_loss)
            w = [w1, w2]
```

这套执行流程就叫作梯度下降（Gradient Descent）。如代码所示，注意在梯度下降的每次迭代中，对  $w_1$  和  $w_2$  是同时更新的。

就这样，如果一切顺利，那么我们可以通过梯度下降，让计算机沿着一条路径搜索到理想的参数组合，进而达成目标——模型的训练。只是这里还有一个小问题，如果在梯度下降的第一步，换一个随机的出发起点又会是怎样呢？

如图 1-17 所示，换一个起点，机器完全可能沿着另一条完全不同的路径，陷入另一个“洼地”，也就是说，梯度下降找到的参数组合只是“局部最优”，并不保证“全局最优”。这个问题从过去的经验上看是可以接受的。比起随机尝试，梯度下降智能地发掘出一条路径，使机器在“合理”的时间内能够找到“足够好”的参数解。

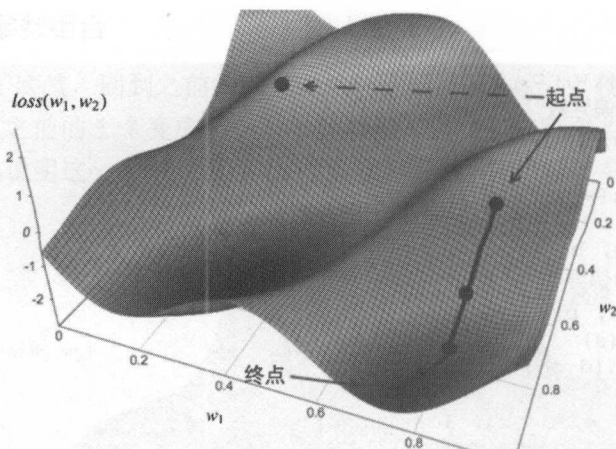


图 1-17 随机另一个起点的梯度下降

### 3. 设置学习速率

在如 `scikit-learn` 等机器学习库中，使用者并不需要注意“学习速率”这一参数，一般机器学习类库会帮你处理好这一参数。有时候出于一些特殊目的，我们需要手动设置学习速率的值。注意，不同的学习速率不仅影响梯度下降收敛的速度和效果，还可能导致收敛失败。为了方便展示，下面在一个简单的函数上观察不同学习速率值对梯度下降的影响。

对于一个简单的二次函数，如果将学习速率设置得过小，则梯度下降收敛的过程会很慢，计算机需要迭代很多次才能找到理想的参数，如图 1-18 所示。

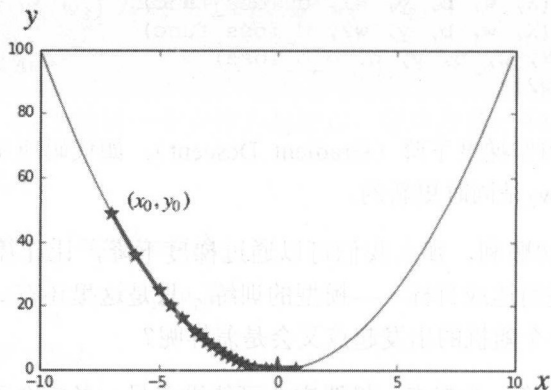


图 1-18 较小的学习速率

反之，如果将学习速率设置得过大，每一步的跨度过长，则导致梯度下降的寻参轨迹将不再收敛，如图 1-19 所示。

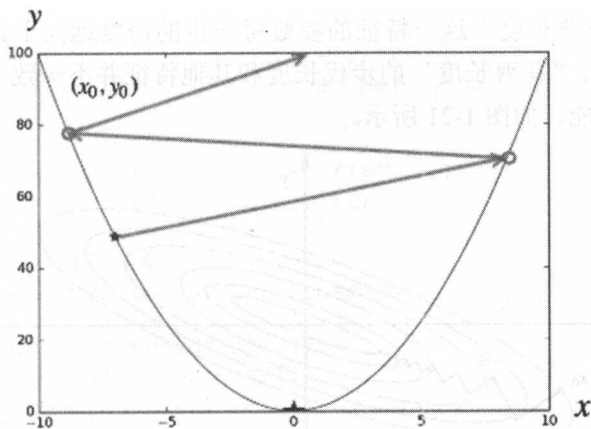


图 1-19 过大的学习速率

### 1.4.2 去均值和归一化

梯度下降给我们带来的不只是模型训练的便捷，还有一些随之而来的数据上的要求。以 IRIS 数据集为例，如果采集的数据大小非常不一致会发生什么？假设花萼长度这一特征按“mm”为单位采集，其他特征长度仍然按“cm”为单位， $x = [300 \ 2 \ 5 \ 3]$ 。

理想中的寻找轨迹每一步都导致  $loss(w, b)$  有效下降，在等高图中看到的效果如图 1-20 所示。

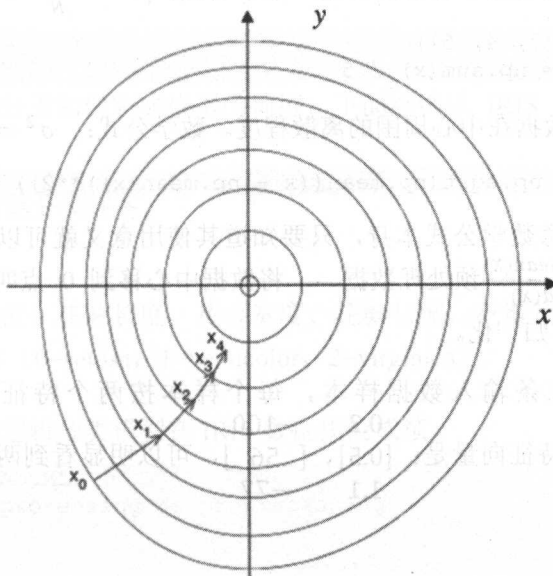


图 1-20 理想的梯度下降

现在，由于“花萼长度”这一特征的参数每一步的跨度远大于其他特征，造成的结果是计算机在寻底时，“花萼长度”的步伐长度和其他特征并不一致，因而会让机器走很多弯路，浪费计算性能。如图 1-21 所示。

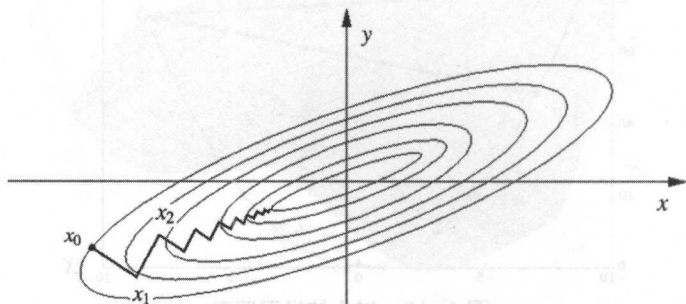


图 1-21 没有归一化的梯度下降

也就是说，梯度下降希望输入的数据在同一尺度上可比较。在 IRIS 的例子中，数据都是在同一单位长度上采集的，因此并没有暴露这一问题，后续章节的例子中我们会碰到这类问题。

解决问题的方法是让输入数据满足均值为 0、方差相似这一统计属性。均值和方差是在数学上用于描述堆在一起的数据表现出来的直观统计属性。

- 均值：描述数据的中心落在哪里。数学公式： $\mu = \frac{\sum_{i=1}^N X_i}{N}$ 。

```
x = np.array([1, 2, 3, 4, 5])
assert np.mean(x) == np.sum(x) / 5
```

- 方差：描述数据在中心周围的离散程度。数学公式： $\sigma^2 = \frac{\sum_{i=1}^n (X_i - \mu)^2}{n}$ 。

```
assert np.std(x) == np.sqrt(np.mean((x - np.mean(x))**2))
```

大家不用过分在意数学公式本身，只要知道其使用意义就可以了。对每个输入特征向量，我们按  $x_i = \frac{x_i - \text{mean}(X)}{\text{std}(X)}$  预处理数据——将数据中心移到 0 点叫作去均值化，将数据范围缩至指定范围叫作归一化。

举个例子，有三条输入数据样本，每个样本按两个特征维度采样，即  $X = \begin{bmatrix} 0.2 & -100 \\ 0.5 & 56 \\ 1.1 & -77 \end{bmatrix}$ ，对应的特征向量是： $[0.5]$ 、 $[56]$ ，可以明显看到两个特征并不在可以比较的尺度上。

归一化特征数据：



```
# 两个特征向量
f1 = np.array([0.2, 0.5, 1.1]).reshape(-1, 1)
f2 = np.array([-100.0, 56.0, -77.0]).reshape(-1, 1)

# 计算归一化
f1_scaled = (f1 - np.mean(f1)) / np.std(f1)
f2_scaled = (f2 - np.mean(f2)) / np.std(f2)

# 用 scikit-learn 封装的函数计算归一化
import sklearn.preprocessing as preprocessing

scaler = preprocessing.StandardScaler()
f1_sk_scaled = scaler.fit_transform(f1)
f2_sk_scaled = scaler.fit_transform(f2)

assert np.allclose(f1_sk_scaled, f1_scaled) and np.allclose(
    f2_sk_scaled, f2_scaled)
```

如代码所示，可以用 `scikit-learn` 的 `preprocessing.StandardScaler` 来快速实现对特征向量的去均值化和归一化。简单来说，去均值化和归一化数据对模型带来的好处无非两点：训练时收敛速度更快以及收敛效果更平稳。归一化会将数据集的数据缩进一定范围，特别地，将数据集缩进  $(0,1)$  范围的手段叫作概率归一化。

除了输入数据集，我们还会对模型的初始随机参数做同样的处理，让其满足均值 0、等方差的分布，防止模型初始参数太过偏离输入数据的量级。

### 1.4.3 实现

前面介绍了逻辑分类如何完成模型的训练，下面继续以 `IRIS` 数据集为例，完成逻辑分类的实现。

问题：`IRIS` 数据集分类任务。

- 150 个样本。
- 四个特征维度：花萼长度、花萼宽度、花瓣长度、花瓣宽度。
- 3-Class 样本（0—`setosa`，1—`versicolor`，2—`virginica`）。

任务目标：获得逻辑分类模型在 `IRIS` 数据集的成绩。

```
from abupy import AbuML
import sklearn.preprocessing as preprocessing

# IRIS 花卉数据集
iris = AbuML.create_test_fiter()
```

```
# 使用逻辑分类, 损失函数指定为交叉熵
iris.estimator.logistic_regression(multi_class='multinomial',
                                   solver='lbfgs')

# cross-validation 测试
iris.cross_val_accuracy_score()
```

输出:

```
accuracy mean: 0.953333333333
array([1., 0.93333333, 1., 0.93333333, 0.93333333,
       0.86666667, 0.93333333, 0.93333333, 1., 1.])
```

## 1.4.4 回顾

- 梯度下降: 通过求导, 寻找通向结果的路径。
- 去均值和归一化: 均值为 0, 方差相似。
  - 原因: 梯度下降要求输入数据整体在同一尺度上方可比较。
  - 效果: 梯度下降收敛更快, 更平稳。

至此, 你已经初步了解了机器学习是什么以及模型训练相关的原理, 下一章将介绍一些进阶的知识。



## 第2章

# 机器学习进阶

本章在第1章基础上，介绍机器学习的进阶知识，包含特征工程、回归模型、模型工程等。

## 2.1 特征工程

You jump, I jump.

——电影《泰坦尼克号》

本节将通过复杂的例子，介绍一些数据、特征的处理技巧。

### 2.1.1 泰坦尼克号生存预测

泰坦尼克号的沉没是历史上最著名的沉船之一。1912年4月15日，在它的首航中，泰坦尼克号在与冰山撞击后沉没，共2224名乘客和船员中丧生了1502人，如图2-1所示。这一骇人听闻的悲剧震撼了国际社会，并为船舶制定了更好的安全条例。海难导致这种生命损失的原因之一是乘客和船员没有足够的救生艇。虽然船员能否生还是有一些运气因素决定的，但整体上，一些人群比其他入更可能生存，如妇女、儿童和上层阶级。



图2-1 泰坦尼克号事故

在这个例子中，我们的任务是完成对什么样的人可能生存的分析，应用机器学习的工具来预测哪些乘客幸免于悲剧。

### 第一步：任务描述

- 样本数：891 名乘客信息（小数据集）
- 原始特征数：11
- PassengerId: 乘客 id
- Pclass: 几等舱(1-一等舱, 2-二等舱, 3-三等舱)
- Name: 名字
- Sex: 性别
- Age: 年龄
- SibSp: 兄弟姐妹/配偶的数量
- Parch: 父母/孩子的数量
- Ticket: 机票号码
- Fare: 票价
- Cabin: 客舱
- Embarked: 登船的港口（C-瑟堡, Q-皇后镇, S-南安普顿）
- 目标：预测 Survived（1-生存, 0-死亡）

### 第二步：观察数据集

说明：从 Kaggle 下载的原始数据集文件包含 train.csv 和 test.csv 两部分。train.csv 用作模型训练，test.csv 的样本没有标签。将模型预测 test.csv 样本的结果提交到 Kaggle 平台后，Kaggle 会给出成绩。本书后面部分出于讲解方便的目的，将忽视 test.csv，直接在 train.csv 分割训练-测试集，显式观测成绩。

我们关心的要点如下：

- 采集的样本是否有缺失数据。
- 不同类别的样本数量分布是否基本均匀。

数据集整体信息：

```
import pandas as pd # pandas 是 Python 的数据格式处理类库

# 加载泰坦尼克号生存预测数据集
data_train = pd.read_csv("../data/titanic/train.csv")
```

```
data_train.info()
```

输出:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
PassengerId    891 non-null int64
Survived       891 non-null int64
Pclass         891 non-null int64
Name           891 non-null object
Sex            891 non-null object
Age           714 non-null float64
SibSp          891 non-null int64
Parch          891 non-null int64
Ticket         891 non-null object
Fare           891 non-null float64
Cabin          204 non-null object
Embarked       889 non-null object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.6+ KB
```

数据缺失程度:

```
data_train.groupby('Survived').count()
```

输出如表 2-1 所示。

表 2-1 titanic 数据集缺失统计

	PassengerId	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
Survived											
0	549	549	549	549	424	549	549	549	549	68	549
1	342	342	342	342	290	342	342	342	342	136	340

可以看到, 样本类别比例为 549:342, 略不均衡, 但差距不大。同时注意到 Age、Cabin、Embarked 维度的数据有缺失, 在开始模型训练之前, 我们需要处理这一情况。

### 第三步: 数据预处理

这里要处理特征的输入数据, 使其适合模型去训练。

首先, 观察并挑选特征。

观察特征:

```
data_train.head(3)
```

输出如表 2-2 所示。

表 2-2 titanic 数据采样

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...)	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S

Survived 是生存预测的数据标签。特征 PassengerId、Ticket、Name 维度下的数据基本是完全随机的，这些特征应该和任务的目标生存预测不相关。

## 2.1.2 两类特征

剩下的输入特征分为两类：一类是有“数值”意义的数据，如年龄 Age、票价 Fare、兄弟姐妹/配偶的数量 SibSp、父母/孩子的数量 Parch；另一类是有“类别”意义的数据，比如几等舱 Pclass、性别 Sex、登船港口 Embarked、客舱 Cabin，这种特征下的数值是没有数量意义的，如特征 Pclass（几等舱）下的样本数值：1 和 3，并不是表示数量或者长度的 3 倍关系，而仅仅是类别标号。

- 数值意义的特征。
- 类别意义的特征。

当这些数据流入模型时，模型并不知道哪些是数量意义的数值，哪些是类别意义的数值，所以我们需要把这两类特征分开处理一下。

### 1. 处理数据缺失

先处理“数值”意义的特征数据，第一个问题是 Age 的数据缺失。补全缺失数据的唯一要点就是：尽量保持原始信息状态。处理手段视情况而定，共有如下几种：

- 扔掉缺失数据。
- 按某个统计量补全；统计量可以是定值、均值、中位数等。
- 拿模型预测缺失值。

对于大数据集的小比例缺失，可以直接扔掉这部分数据样本，因为样本充分，信息不会损失多少。但泰坦尼克号数据样本总共 891 条，每条数据样本对模型的训练都是珍贵的，所以我们希望补全这部分信息而不是丢弃。在填补 Age 的缺失值时，明显拿模型预测缺失值更好。因为直观上，我们可以猜测 Pclass、Fare、Parch、SibSp 等特征是和上船的年龄相关的特征。我们完全可以拿一种回归预测模型，输入这些特征，预测年龄，这样新增的数据就是基于原始数据信息的合理猜测。由于回归模型在本书 2.4 节才登场，因而这里先用均值这一手段填补缺失数据。

均值填充：

```
def set_missing_ages(p_df):
    p_df.loc[(p_df.Age.isnull()), 'Age'] = p_df.Age.dropna().mean()
    return p_df
df = set_missing_ages(data_train)
```

## 2. 归一化数值数据

对于两类特征数据混杂的问题，我们可以看到，票价 Fare、年龄 Age 这两个数值特征和其他特征明显不在同一可以比较的尺度上。回顾 1.3.6 节所说的梯度下降对输入数据的要求，我们希望所有的输入数据在差不多同一尺度上可比较，所以下面归一化这两个维度的数据。

归一化数据：

```
import sklearn.preprocessing as preprocessing

scaler = preprocessing.StandardScaler()
df['Age_scaled'] = scaler.fit_transform(data_train['Age'])
df['Fare_scaled'] = scaler.fit_transform(data_train['Fare'])
```

## 3. 处理类别意义的特征

客舱号 Cabin 的缺失程度太严重了，在整体样本很小， $1 - \frac{204}{891} = 77\%$  的缺失程度的数据不可能有效恢复。不过虽然客舱号信息 Cabin 无法提取，但可以将“有没有在客舱”这一信息提取出来。

处理 Cabin 特征：

```
def set_cabin_type(p_df):
    p_df.loc[(p_df.Cabin.notnull()), 'Cabin'] = "Yes"
    p_df.loc[(p_df.Cabin.isnull()), 'Cabin'] = "No"
    return p_df
df = set_cabin_type(df)
```

对于类别意义的特征，数值大小没有任何数量上的意义。就像前面提到的，对于

Pclass, 1 和 3 并不是表示数量关系, 而是类别标号。对于类别标号有意义的只有“是”这一类和“不是”这一类。所以, 对于所有的类别意义的特征, 下面将按类别标号重新建立新的特征, 特征的数值只有 1 和 0, 标识样本“是”这一类或者“不是”这一类。

处理 Pclass 特征:

```
dummies_pclass = pd.get_dummies(data_train['Pclass'], prefix='Pclass')
dummies_pclass.head(3)
```

输出如表 2-3 所示。

表 2-3 Pclass 特征数据采样

Pclass_1	Pclass_2	Pclass_3
0.0	0.0	1.0
1.0	0.0	0.0
0.0	0.0	1.0

对于特征 Embarked, 缺失的数据样本只有两条, 对于这两条数据, 在按特征标签展开时, 每个特征标签数值都为 0。

处理 Embarked 特征:

```
dummies_embarked = pd.get_dummies(data_train['Embarked'],
                                   prefix='Embarked')
dummies_embarked.loc[61]
```

输出:

```
Embarked_C    0.0
Embarked_Q    0.0
Embarked_S    0.0
Name: 61, dtype: float64
```

预处理其他类别数据, 把 Sex 的文本数据换成数字类别标号:

```
dummies_sex = pd.get_dummies(data_train['Sex'], prefix='Sex')
dummies_sex.head(3)
```

输出如表 2-4 所示。

表 2-4 Sex 特征数据采样

Sex_female	Sex_male
0.0	1.0
1.0	0.0
1.0	0.0



接下来把处理好的数据维度合并进去，把不需要的数据维度扔掉：

```
df = pd.concat([df, dummies_embarked, dummies_sex, dummies_pclass],
               axis=1)

# noinspection PyUnresolvedReferences
df.drop(['Pclass', 'Name', 'Sex', 'Ticket', 'Cabin', 'Embarked'],
        axis=1, inplace=True)
```

至此，输入数据就预处理好了，看下模型将要用哪些特征：

```
# 选择哪些特征作为训练特征
train_df = df.filter(regex='Survived|Age_|SibSp|Parch|Fare_|'\
|Cabin_|Embarked_|Sex_|Pclass_|')
train_df.head(1)
```

输出如表 2-5 所示。

表 2-5 模型选用的特征

	Survived	SibSp	Parch	Age_scaled	Fare_scaled	Embarked_C	Embarked_Q	Embarked_S	Sex_male	Sex_male	Pclass_1	Pclass_2	Pclass_3
0	0	1	0	-0.592	-0.502	0.0	0.0	1.0	0.0	1.0	0.0	0.0	1.0

接入模型看看成绩：

```
from abupy import AbuML

train_np = train_df.as_matrix()
y = train_np[:, 0]
x = train_np[:, 1:]
titanic = AbuML(x, y, train_df)

titanic.estimator.logistic_regression()
titanic.cross_val_accuracy_score()
```

输出：

```
accuracy mean: 0.79919731018
array([0.77777778, 0.78888889, 0.7752809, 0.82022472, 0.78651685,
       0.7752809, 0.78651685, 0.79775281, 0.83146067, 0.85227273])
```

### 2.1.3 构造非线性特征

我们希望逻辑分类模型可以做得更好。前面提到，逻辑分类是一个“线性模型”，所谓线性模型就是把特征对分类结果的作用加起来，也就是说线性模型能表示类似于  $y = x_1 + x_2$  关系的表达式（ $y$  表示分类结果， $x_1$ 、 $x_2$  表示特征对分类的作用），但线性模型无法表示一些非线性的关系如  $y = x_1 \cdot x_2$ 。所以我们打算人工构造一些新的特征，弥补线性模型对非线性表达式表达能力的不足。

特征的非线性的表达式可以分为两类：

(1) 用于表达“数值特征”本身的非线性因素。

(2) 用于表达特征与特征之间存在非线性关联，并且这种关联关系对分类结果有帮助。

第一种情况是说特征对目标类别的作用不是线性关系。比如两个样本的特征数值是 1 和 3，对应地，这个特征对分类产生的作用其实是  $1^2$  和  $3^2$ ，或者  $\ln 1$  和  $\ln 3$ 。这类问题的本质是数字内在的线性数量描述并不符合真实的关系描述。

对于第一种，仅适用于数值特征，对应的构造特征的方式有两种：多项式化和离散化。多项式构造指的是将原有数值的高次方作为特征；数据离散化是指将连续的数值划分成一个个区间，以数值是否在区间内作为特征。高次方让数值内在表达变得复杂，可描述能力增强；而离散则是让模型来拟合逼近真实的关系描述。

举个简单的例子，对于特征 Age，可以构造平方特征，也可以拿是否满足  $Age \leq 10$  这一条件划分区间构造出新特征。

```
# 划分区间
df['Child'] = (data_train['Age'] <= 10).astype(int)
# 平方
df['Age*Age'] = data_train['Age'] * data_train['Age']
# 归一化
df['Age*Age_scaled'] = scaler.fit_transform(df['Age*Age'])
```

接着尝试构造新特征表达特征与特征之间的非线性关联，同样源自多项式的思路。比如，我们觉得 Pclass 数值越大越不容易生存下来，头等舱的遇害人员应该比三等舱的更可能被照顾；同时年龄越大的人也越不容易生存下来，越小的越可能被照顾，这两个特征之间会不会也有一些关联，并且这种关联对生存预测有指导意义呢？

我们可以构造一个新特征 “Age \* Class”，加入模型中。

```
df['Age*Class'] = data_train['Age'] * data_train['Pclass']
# 归一化
df['Age*Class_scaled'] = scaler.fit_transform(df['Age*Class'])
```

看下模型现在用的特征：

```
# filter 加入新增的特征
train_df = df.filter(regex='Survived|Age_|SibSp|Parch|Fare_|'\
|Cabin_|Embarked_|Sex_|Pclass_|Child|Age*Class_')
train_df.head(1)
```



输出如表 2-6 所示。

表 2-6 模型选用的特征

	Survived	SibSp	Parch	Age_scaled	Fare_scaled	Embarked_C	Embarked_Q	Embarked_S	Sex_female	Sex_male	Pclass_1	Pclass_2	Pclass_3	Chi_sqld	Age*Age_scaled	Age*Class_scaled
0	0	1	0	-0.592	-0.502	0.0	0.0	1.0	0.0	1.0	0.0	0.0	1.0	0	-0.636573	0.031376

新加入的这些特征，对模型的表现是否有提升呢？

```
train_np = train_df.as_matrix()
y = train_np[:, 0]
x = train_np[:, 1:]
titanic = AbuML(x, y, train_df)

titanic.estimator.logistic_regression()
titanic.cross_val_accuracy_score()
```

输出：

```
accuracy mean: 0.804752298264
array([0.8, 0.8, 0.78651685, 0.84269663, 0.79775281,
       0.76404494, 0.79775281, 0.78651685, 0.84269663, 0.82954545])
```

## 1. 评估特征作用

一般而言，机器学习中看一个新特征是否发挥作用，最常用的方法就是加进去看模型成绩是否提升。可以同时观察模型给特征分配的权重，看特征发挥作用的大小。

```
titanic.importances_coef_pd()
```

输出如表 2-7 所示。

表 2-7 模型特征权重

	coef	columns
0	[-0.410298136384]	SibSp
1	[-0.173336882199]	Parch
2	[-0.208459115049]	Age_scaled
3	[0.165577117349]	Fare_scaled
4	[0.0]	Embarked_C
5	[0.0]	Embarked_Q
6	[-0.405822408349]	Embarked_S

续表

	coef	columns
7	[1.99640288087]	Sex_female
8	[-0.715926302128]	Sex_male
9	[0.663404977168]	Pclass_1
10	[0.0]	Pclass_2
11	[-1.03255600599]	Pclass_3
12	[1.28640364062]	Child
13	[0.0]	Age*Age_scaled
14	[-0.129278692295]	Age*Class_scaled

可以看到，在训练好的模型中，特征 Child 有效发挥了作用，而 Age\*Class、Age\*Age 没有什么用。

对于一些特定的应用场景中，模型的训练非常耗时，可能几天甚至更长。这些应用场景中，需要一些新的数学方法估计新特征是否有效。机器学习中有许多方法评估特征在模型中发挥的作用，对于这些方法，你只需知道它们基本上都是通过某种数学公式，计算特征和预测值之间的相关性就可以了。如 RFE selection。

```
titanic.feature_selection()
```

输出：

```
RFE selection
           ranking support
SibSp           1    True
Parch           3   False
Age_scaled       2   False
Fare_scaled      4   False
Embarked_C       9   False
Embarked_Q       8   False
Embarked_S       1    True
Sex_female       1    True
Sex_male         1    True
Pclass_1         1    True
Pclass_2         7   False
Pclass_3         1    True
Child            1    True
Age*Age_scaled   6   False
Age*Class_scaled 5   False
```

## 2. 构造特征的数学意义

这里尝试讲得稍微深入一点。通过人工构造非线性特征，可以弥补线性模型表达能力的不足。这一手段之所以能够生效，背后的原因是：低维的非线性关系可以在高维空间

线性展开。

解释这一观点，让我想起霍金先生的巨著《时间简史》中提及的一个例子：这正如同看一架在非常多山的地面上空飞行的飞机。虽然它沿着三维空间的直线飞，在二维的地面上它的影子却是沿着一条弯曲的路径——飞机影子的运动轨迹在二维地表上看到的是一条非线性的曲线，起伏不定，很难用函数表示，如图 2-3 所示；但在三维空间中，其运动轨迹仅仅是一条笔直的直线而已，一个简单的线性函数就可以说明。也就是说，在二维空间中看到的复杂表达式，当增加一个维度后，其表达式可能就变得非常简单了。



图 2-2 飞机运动轨迹的投影

同样的道理，我们可以增加新的特征维度，让分类任务背后的数学表达式变得更简单，让分类模型更容易挖掘出信息——这正是构造新特征有意义的地方：增加特征维度，构造出模型表达不出来的内在表达式。对于逻辑分类模型而言，就是通过增加新的非线性特征，完成特征维度的扩展，构造出模型表达不出的非线性的内在关系。

逻辑分类因为是线性模型，原型简单，所以有着训练速度快、易分布式部署等特点。现在业界对逻辑分类仍然有着广泛的应用，尤其适合那些数据海量、特征高维并且稀疏的应用场景。比如在一些涉及海量用户的个性化推荐任务中，海量数据上，把每个用户 ID 作为一个特征，使用逻辑分类就可以快速有效地完成任务目标。

## 2.1.4 回顾

- 两种特征数据：数值特征数据、类别特征数据。
- 数据预处理：
  - 处理数据缺失。
  - 归一化数值特征数据。

— 按特征标签展开类别特征数据。

- 构造新特征的数学意义：增加特征维度，构造出模型表达不出来的内在表达式。

## 2.2 调试模型

当局者迷，过犹不及。

——裘椹双树《浮生物语》

前几小节中，我们一直在关注模型的训练问题，本节谈谈如何调试模型。

2.1 节讲到可以通过构造新的特征，弥补模型能力的不足，并介绍了两种构造方式：多项式和区间划分。理论上我们可以构造无限多的特征，比如对于“泰坦尼克号生存预测”的例子，仅仅按“Age”和“PClass”的多项式扩展，以多项式  $(Age + PClass)^n = Age^n + nAge^{n-1}PClass^1 + n(n-1)Age^{n-2}PClass^2 \dots$  可以扩展出无限多的特征。然而是不是模型的特征真的越多越好？特征维度究竟扩展到什么程度才是合理的？

### 2.2.1 模型调试的目标

#### 1. 过拟合

在实际中我们发现，当特征维度增加到一定程度时，虽然模型在训练集的成绩越来越好，但在测试集的最终成绩反而越来越差。模型分类在训练集的表现行为可以用图 2-3 说明。

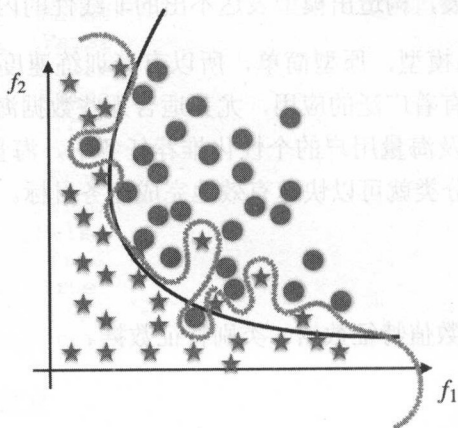


图 2-3 过拟合

图 2-3 将数据集样本画在平面图上，黑色的实线代表我们期望的分类边界线（视图中这种分类边界线叫作“判定边界”），红色则是过多的特征下模型的表现。模型对训练集的过度拟合就叫作过拟合。

造成这一现象的原因是模型训练过程中，每个特征都仅仅是让模型努力在“已知”的训练集数据上做得更好。由于本身训练集的数据和测试集并不可能完全一样，会有一些特性差异的（这个差异在小数据下格外明显）。让模型在训练集数据表现得过分好，就会导致模型在测试集上“不适应”。也就是说：模型对数据是有记忆的。这就像一个学生，在学习的时候过分地记住了训练集每道题的答案，却没有学习到一个“通用”的解题方法。

解决过拟合问题的方法有很多。一种常用的方式是通过在惩罚函数中新增加一个正则化参数  $C$  来控制分类边界对样本的辨识度，用新的参数乘以权重矩阵的平方  $\frac{w^T w}{C}$ ，对于逻辑分类，新的惩罚函数变为  $loss(w, b) = -\frac{1}{M} \sum_i D(y_i, p_i) + \frac{1}{M} \cdot \frac{w^T w}{C}$ 。由于我们用的是权重的二次方，所以这种正则化方式叫作 L2 正则化。如果使用  $\frac{|w|}{C}$  正则，则对应叫作 L1 正则化。

L2 的损失函数：

```
# L2 正则化
# X 是训练样本矩阵，W 是权重矩阵，b 是偏置向量，y 是真实标签矩阵
def loss_func(X, W, b, y):
    # lmd 是一个可调节的模型参数
    C = 2
    s = score(X, W, b)
    p = softmax(s)
    return -np.mean(cross_entropy(y, p)) + np.mean(np.dot(w.T, w) / C)
```

图形上，可以理解这个新增的参数  $C$  就是在控制判定边界线的“柔和度”， $C$  越小， $\frac{w^T w}{C}$  越大，判定边界线越“柔和”，对训练数据以外的未知数据越适应。这样做的最大好处是，解决模型的过拟合问题变成了处理模型参数的调试问题。

还有很多其他的解决过拟合的手段，比如增加更多的训练数据。训练数据集越大，模型越不容易对单个样本过度“记忆”。以泰坦尼克号生存预测问题为例，横坐标是训练集数据量，纵坐标是预测成绩，可以将模型的学习曲线画出来，如图 2-4 所示。

学习曲线：

```
from abupy import AbuML
```



```
# create_test_more_fiter 用于快速测试 API, 封装了 titanic 数据集合相关的特征处理
titanic = AbuML.create_test_more_fiter()
titanic.estimator.logistic_regression()

# 学习曲线
titanic.plot_learning_curve()
```

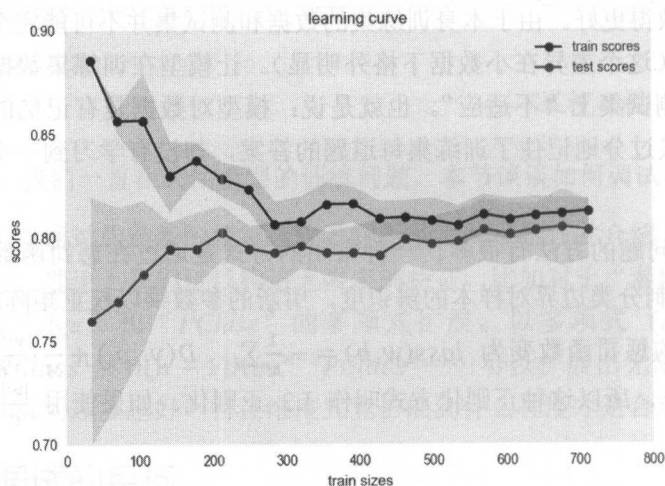


图 2-4 学习曲线

备注：彩图见随书 Git 库示例代码运行结果。

可以看到，随着训练样本数量的增加，训练集成绩降低，测试集成绩升高，最后两者趋于平稳，测试集成绩略低于训练集。图 2-4 是一个合理训练的模型例子，在发生过拟合的模型学习曲线中，训练集成绩非常高，甚至满分，而相反测试集成绩会非常低。

小结一下，处理过拟合的方法是：

- 减少特征，降低模型复杂度。
- 减小调试参数  $C$ 。
- 增加训练数据量。

## 2. 欠拟合

与过拟合相对的是模型的欠拟合问题，欠拟合就是模型能力不足，没有有效地表示出数据集的内在表达式，如图 2-5 所示。

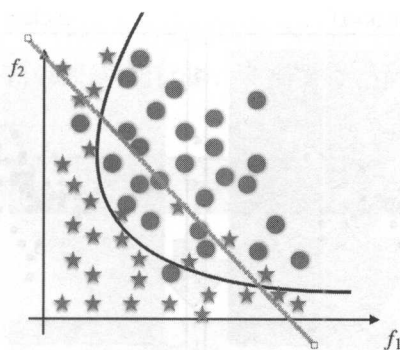


图 2-5 欠拟合

处理欠拟合的手段很简单：

- 增加特征，增大模型复杂度。
- 增大调试参数  $C$ 。

总之，模型越简单，模型适应力越强，越容易发生欠拟合；反之，模型越复杂，模型适应力越差，越容易发生过拟合。

如图 2-6 所示，调试模型的最大关键就是寻找过拟合和欠拟合的平衡。

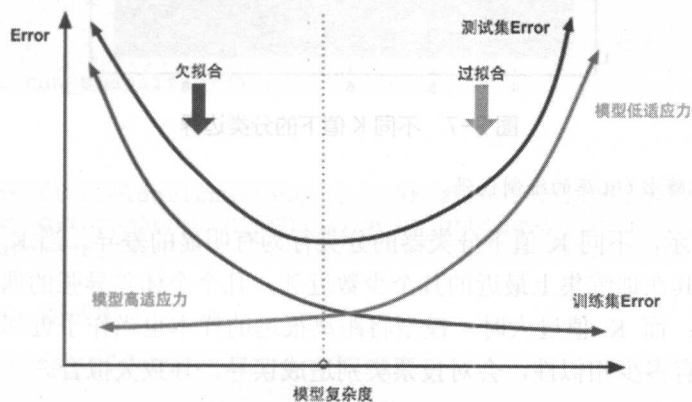


图 2-6 平衡过拟合和欠拟合

## 2.2.2 调试模型

### 1. 待调试的模型参数

在前面的小节中，我们介绍了两种模型：KNN 和逻辑分类。每个模型都有一些参数需要调节，这些参数会直接影响到模型训练速度、分类的效果，等等。对于 KNN，需要调节的参数是  $K$  值（近邻的数量），不同  $K$  值下的分类边界如图 2-7 所示。



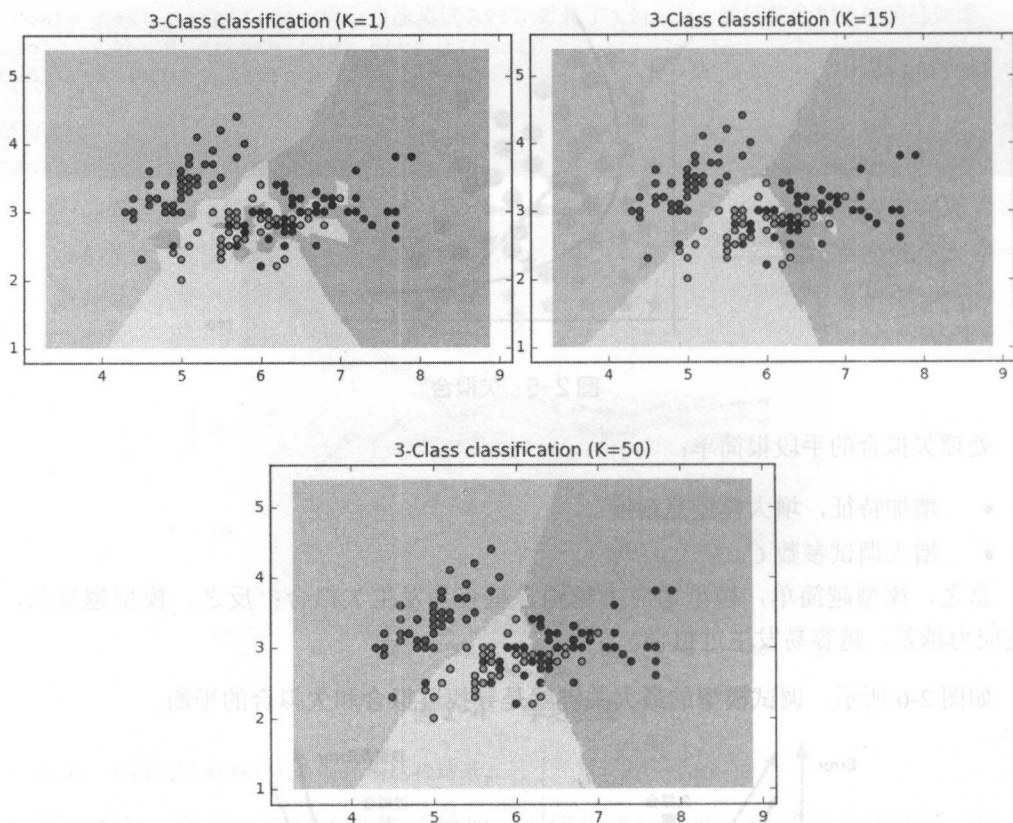


图 2-7 不同 K 值下的分类边界

备注：彩图见随书 Git 库的绘制代码。

如图 2-7 所示，不同 K 值下分类器的分类行为有明显的差异。当 K 值很小时，待测样本的分类依赖其在训练集上最近的几个少数近邻，几个个体差异强的偶然近邻足以导致分类器分类失败；而 K 值过大时，模型将距离很远的样本也当作了近邻，这些样本与待测样本类别并没有多少相似性，会对投票类别造成误导，导致欠拟合。

对于逻辑分类，需要调节的参数有：用来解决过拟合、欠拟合的参数 C。对于不同模型的这些参数，我们希望找到某一种通用的方式来完成调试。

## 2. 交叉验证 (Cross-validation)

调试模型参数的思路很简单：在训练数据上训练好模型，在测试数据看成绩，将测试集上成绩最好的参数组合作为模型参数。这种思路叫作交叉验证。

之前我们提到：模型对数据是有记忆的。为了防止固定的数据分割方式造成模型依

旧对特定的数据集过度记忆，一种解决思路叫作 N-fold Cross-validation。实现方法很简单，依次对数据集不同部分划分训练-测试集，将所有成绩的平均值作为当前模型参数的成绩。如图 2-8 所示。

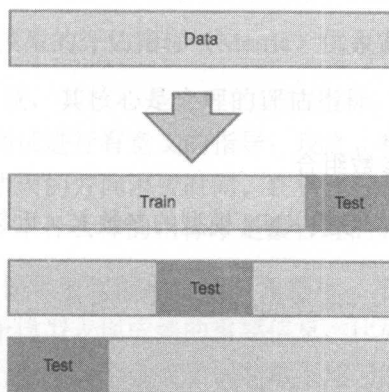


图 2-8 N-fold Cross validation

使用 KNN:

```
# IRIS 花卉数据集
iris = AbuML.create_test_fiter()

# 使用 KNN
iris.estimator.knn_classifier()
```

输出:

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
metric_params=None, n_jobs=1, n_neighbors=1, p=2,
weights='uniform')
```

分割训练集、测试集:

```
from sklearn.cross_validation import KFold

kf = KFold(len(iris.y), n_splits=10, shuffle=True)

for train_index, test_index in kf:
    x_train, x_test = iris.x[train_index], iris.x[test_index]
    y_train, y_test = iris.y[train_index], iris.y[test_index]

x_train.shape, x_test.shape, y_train.shape, y_test.shape
```

输出:

```
((135, 4), (15, 4), (135,), (15,))
```

### 3. GridSearch

GridSearch 是在 N-fold Cross-validation 基础上的封装实现。可以通过设置一个参数搜索空间，暴力搜索所有参数组合，可以同时寻找多个最优参数。

寻找最佳参数组合只需两步：

- (1) 定义参数搜索范围
- (2) 在数据中尝试所有参数组合

如 IRIS 花卉分类问题中，寻找 KNN 模型的参数 K（近邻数量）的最佳值：

```
# 定义参数搜索范围
param_grid = dict(n_neighbors=range(1, 31))

# grid search
best_score_, best_params_ = iris.grid_search_common_clf(param_grid,
                                                         cv=10, scoring='accuracy')
best_score_, best_params_
```

输出：

```
(0.9799999999999999, {'n_neighbors': 13})
```

在泰坦尼克号生存预测问题中，用来调整逻辑分类模型的过拟合/欠拟合平衡的参数 C，查找如下：

```
# 定义参数搜索范围
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000]}

# grid search
best_score_, best_params_ = titanic.grid_search_common_clf(param_grid,
                                                            scoring='accuracy')
best_score_, best_params_
```

输出：

```
(0.80246913580246915, {'C': 1})
```

GridSearchCV 是目前使用最广泛的最优参数搜索方法。

## 2.2.3 回顾

- 模型对数据是有“记忆”的。
- 调试模型的主要目标：平衡过拟合和欠拟合。
- 通过 GridSearch 暴力搜索最佳的模型参数组合。

## 2.3 分类模型评估指标

人生是一个万花筒，换不同的角度去观察，你会发现全新的世界。

本节将介绍更多的分类模型的评估指标（Metric）供大家在使用时选择。

一个好的机器学习系统，其核心是合理的评估指标。对于任务目标而言，只有好的评估才能对模型的选择、调试进行有意义的指导；反之，糟糕的或者不合适的评估方式会将开发者带入歧途，朝着错误的方向浪费时间。在搭建任何机器学习系统之前，第一步都应该先去反思业务，思考一下什么样的指标才是最合理的。

到目前为止，我们评估模型表现的成绩就一条指标：准确率。然而准确率的计算方式过于简单，缺失了很多关于模型表现成绩的重要信息，比如，模型预测样本的分布、错误预测的类型等。举个例子，想象一个分类类别非常不均匀的数据集，如癌症病例预测：正常样本占比 99%，患者样本占比 1%。现有两个模型：模型 A 经过复杂的表达式及调试，训练成绩是准确率 95%；模型 B 则无论输入样本是什么，总是预测正常，所以准确率是 99%。虽然从准确率的成绩上看 B 比 A 表现更好，但从实际使用上很难说 B 比 A 更优秀，因为发现那 1% 的患者才是这个模型的任务目标——在“类别不均衡”的数据集中，准确率失效。所以我们希望获得一些新的指标，可以多方面地评估模型的表现。

### 2.3.1 混淆矩阵系指标

我们打算先获得模型预测样本成绩的分布，再从中扩展出多种评估指标。这样的评估方式可以帮助我们知道模型究竟在哪些局部样本表现得不够好。很多指标都是基于二分类评估的指标，所谓二分类，就是它们只评估模型针对某一种类别辨别的能力，其中标签为 1 的样本表示“是”这一类别，叫作“正样本”，标签为 0 的样本表示“不是”这一类别，叫作“负样本”。当然，这些二分类指标也可以扩展到多分类上，详见下文。

#### 1. 混淆矩阵（Confusion Matrix）

混淆矩阵将预测标签数量和真实标签数量进行统计，放入矩阵表格中，它的物理意义在于观察模型对样本预测的成绩的数量分布。以泰坦尼克号生存预测为例（生存：1，死亡：0），模型的预测标签与真实标签之间的分布关系，按混淆矩阵的计算方式分为 4 部分。

- **TP**（True Positives）：模型预测为 1，并且真实样本也为 1。
- **TN**（True Negatives）：模型预测为 0，并且真实样本也为 0。

- **FP** (False Positives): 模型预测为 1, 但真实样本为 0。
- **FN** (False Negatives): 模型预测为 0, 但真实样本为 1。

如图 2-9 所示。

	预测: 0	预测: 1
真实: 0	TN = 448	FP = 101
真实: 1	FN = 98	TP = 244

图 2-9 混淆矩阵—数字示意图

总样本数  $N = 448 + 101 + 98 + 244 = 891$ 。可以用一行代码画出混淆矩阵:

```
titanic.plot_confusion_matrices()
```

输出:

```
[[444 105]
 [ 87 255]]
```

如图 2-10 所示。

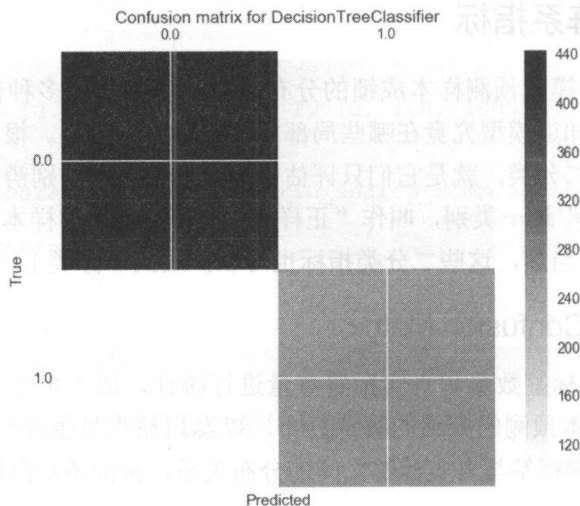


图 2-10 混淆矩阵—颜色图

不同深度的颜色块代表不同的数值。展开代码的细节, 关于 **TP**、**TN**、**FP**、**FN** 的计



算如下:

```
from abupy import ABuMLExecute
from sklearn import metrics

titanic_y_pred = ABuMLExecute.run_cv_estimator(
    titanic.get_fiter(), titanic.x, titanic.y, n_folds=10)
confusion_matrix = metrics.confusion_matrix(titanic.y, titanic_y_pred)
TP = confusion_matrix[1, 1]
TN = confusion_matrix[0, 0]
FP = confusion_matrix[0, 1]
FN = confusion_matrix[1, 0]
TP, TN, FP, FN
```

输出:

```
(245, 455, 94, 97)
```

由混淆矩阵可以衍生出很多数值指标, 常用的有:

- 准确率 (Accuracy)。
- 精确率和召回率 (Precision-Recall)。
- F1 分数 (F1-Score)。

## 2. 准确率 (Accuracy)

之前使用的准确率其实就是模型正确分类标签的比例:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

准确率可以在类别均匀的样本数据中, 对模型整体的分类能力做评估。

```
assert metrics.accuracy_score(titanic.y, titanic_y_pred) == \
    (TP + TN) / float(TP + TN + FP + FN)
```

## 3. 精确率和召回率 (Precision-Recall)

精确率和召回率是二分类指标, 取值在 0 和 1 之间, 通常组合在一起发挥作用。精确率评估模型的预测数据中正样本的准确率:

$$Precision = \frac{TP}{TP + FP}$$

召回率评估模型的预测数据中正样本的覆盖率:

$$Recall = \frac{TP}{TP + FN}$$



通俗地讲，精确率就是描述当模型说样本为“是”这一类别时，可信的程度；而召回率则描述一种覆盖率，表示模型的预测能抓住这一类别的样本占这一类别全部样本的比例。

精确率和召回率：

```
# “生存”类别的精确率
tit_precision = TP / float(TP + FP)
# “生存”类别的召回率
tit_recall = TP / float(TP + FN)

assert metrics.precision_score(titanic.y,
                               titanic.y_pred) == tit_precision
assert metrics.recall_score(titanic.y, titanic.y_pred) == tit_recall
```

我们希望准确率和召回率都越高越好，但事实上，这两者在某些情况下是有矛盾的。比如在泰坦尼克号生存预测中，极端情况下，只预测一个样本生还，并且是准确的，那么精确率就是 1，但是召回率就会非常低；而如果预测全部样本都生还，那么召回率是 1，但精确率就很低。因此在不同的场合下我们需要取舍，是希望精确率比较高或是召回率比较高。比如像疾病预测等分类场景中，精确率就比召回率会更重要些；而像检索、推荐中（类别标签：“是”或“不是”用户喜欢的主题），当返回正样本数量较小时，召回率肯定比精确率更优先。

#### 4. F1 分数 (F1-Score)

在一些场景中，如果精确率和召回率几乎同等重要，那么我们就可以拿 F1 分数评估模型的表现，F1 分数值就是精确率和召回率的倒数平均数。

$$F1 = \frac{1}{\frac{1}{Precision} + \frac{1}{Recall}} = \frac{2 * Precision * Recall}{Precision + Recall}$$

F1 分数：

```
assert metrics.f1_score(titanic.y, titanic.y_pred) == \
    2 * tit_precision * tit_recall / (tit_precision + tit_recall)
```

#### 5. 多分类的指标

二分类的评估指标也可以扩展到多分类问题中，方法是将每个多分类问题看作多个二分类问题分开评估。以 IRIS 花卉多分类为例，模型的任务目标（0-setosa, 1-versicolor, 2-virginica）分成三个二分类任务（1-setosa、0-not setosa）、（1-versicolor、0-not versicolor）、（1-virginica、0-not virginica）分开评估。

```
# IRIS 花卉数据集
```

```
iris = AbuML.create_test_fiter()
iris.estimator.knn_classifier(n_neighbors=20)
```

输出:

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=1, n_neighbors=20, p=2,
                     weights='uniform')
```

画出来:

```
confusion = iris.plot_confusion_matrices()
```

输出:

```
[[50  0  0]
 [ 0 48  2]
 [ 0  1 49]]
```

如图 2-11 所示。

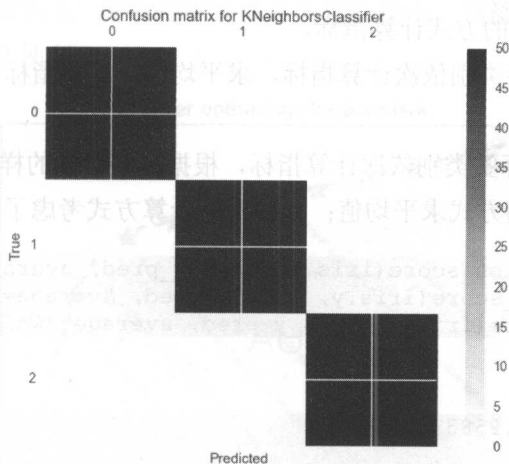


图 2-11 混淆矩阵—多分类

其中, 按标签计算每个类别的精确率/召回率/ $F1$  分数:

输入:

```
iris_y_pred = ABuMLExecute.run_cv_estimator(iris.get_fiter(), iris.x,
                                             iris.y, n_folds=10)
# precision
metrics.precision_score(iris.y, iris_y_pred, average=None)
```

输出:

```
array([ 1. , 0.92307692, 0.95833333])
```

输入:

```
# recall
metrics.recall_score(iris.y, iris_y_pred, average=None)
```

输出:

```
array([ 1. , 0.96, 0.92])
```

输入:

```
# f1-score
metrics.f1_score(iris.y, iris_y_pred, average=None)
```

输出:

```
array([ 1. , 0.94117647, 0.93877551])
```

将几个分类的指标合并起来，可以指定三种合并方式。

- **micro**: 按全局的方式计算指标。
- **macro**: 按标签类别依次计算指标，求平均值；这种指标计算方式无视样本类别不均衡。
- **weighted**: 按标签类别依次计算指标，根据标签类别的样本数量计算权重，按指标\*数量权重的方式求平均值；这种指标计算方式考虑了样本类别不均衡。

```
print metrics.precision_score(iris.y, iris_y_pred, average=None)
print metrics.recall_score(iris.y, iris_y_pred, average='macro')
print metrics.f1_score(iris.y, iris_y_pred, average='weighted')
```

输出:

```
[ 1.  0.92307692 0.95833333]
0.96
0.959983993597
```

## 2.3.2 评估曲线

### 1. ROC 曲线

ROC 曲线是一种评估模型分类能力的曲线指标，用于二分类场合下。曲线的画法略复杂：在二分类的场合中，分类器输出的概率按阈值（模型输出  $P > shreshold$  标签为 1,  $P \leq shreshold$  标签为 0）判断样本标签。ROC 曲线中纵坐标是召回率：正确地判断为正样本占全部的比率（True Positive Rate），横坐标则是错误地判断为正样本占全部的比率（False Positive Rate），ROC 按阈值从 1 到 0 递减，将模型对应的召回率数值点连起来。

比如阈值—召回率对应的数据如表 2-8 所示。

表 2-8 阈值—召回率数据

标 号	阈 值	召回率
1	1	0
2	0.9	0.52
3	0.8	0.68
4	0.7	0.72
5	0.6	0.76
6	0.5	0.8
7	0.4	0.84
8	0.3	0.88
9	0.2	0.92
10	0.1	0.96

阈值为 1 时，召回率为 0，模型没有发现该类别的样本。当阈值为 0 时，该类别的样本全部被召回，如图 2-12 所示。

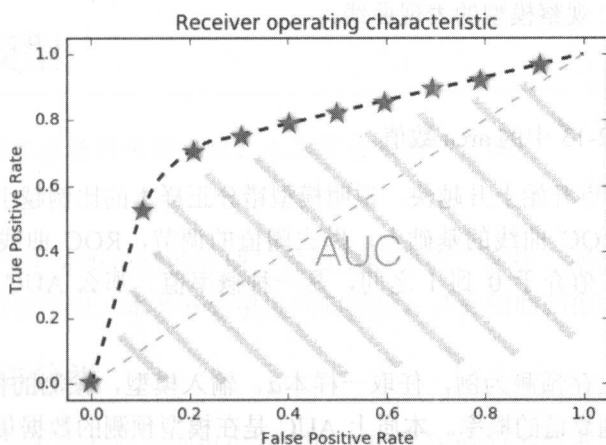


图 2-12 ROC 曲线

下面用代码看看在不同数据集片段上的 ROC 曲线：

```
# K-folder 下的 ROC 曲线和 AUC 面积 (area)
titanic.plot_roc_estimator()
```

输出：

```
DecisionTreeClassifier :roc
```

如图 2-13 所示。

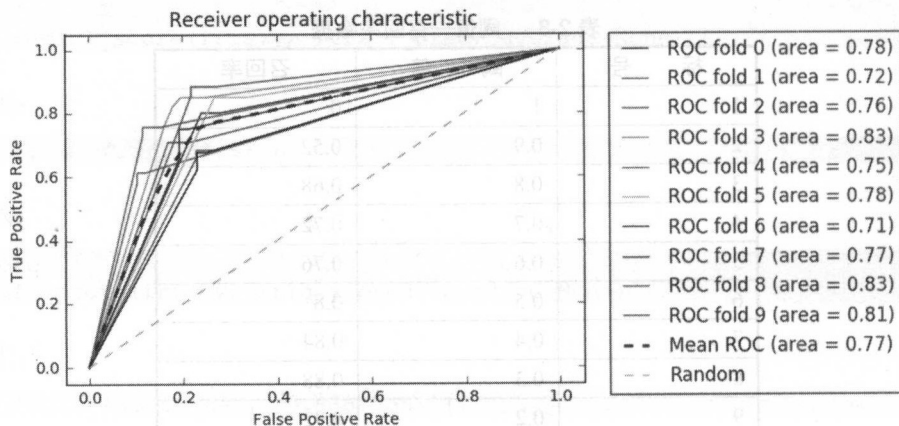


图 2-13 不同数据集片段上的 ROC 曲线

ROC 就是观察模型在不同阈值下，召回率的变化。它有意义的地方在于：当测试集中的正负样本的分布变化的时候，ROC 曲线能够保持不变。因此在类别不均衡的数据集中，也可以使用 ROC 观察模型的表现曲线。

## 2. AUC 指标

AUC 数值即图 2-13 中的 area 数值。

ROC 曲线从 0 点开始上升越快，说明模型错分正样本的比例越小，模型对正样本识别的能力越强。在 ROC 曲线的基础上，抛去阈值的调节，ROC 曲线下半部分的面积值就是 AUC 值。AUC 值介于 0 到 1 之间，是一种概率值。那么 AUC 面积究竟有什么作用呢？

以泰坦尼克号生存预测为例，任取一样本 $a$ ，输入模型，预测的标签为 1（生存），AUC 值就是这一预测靠谱的概率。本质上 AUC 是在模型预测的数据集中，比较正负样本，评估正样本分数排在负样本之上的能力，进而估计模型对正样本预测的可信程度。

由于 AUC 指标能较好地概括不平衡类别的样本集下分类器的性能，因此成为很多机器学习系统中的最终判定标准。当然，在实际使用时，我们还是要根据具体任务选择指标，特殊的任务甚至需要自己定制合适的指标函数。分类评估指标如表 2-9 所示。

表 2-9 分类评估指标

分类评估指标	scikit-learn 模块	内 容
accuracy	metrics.accuracy_score	准确率，简单，最常用
precision	metrics.precision_score	精确率，常与召回率组合使用
recall	metrics.recall_score	召回率，常与精确率组合使用
f1	metrics.f1_score	F1-分数，用于二分类
f1_micro/f1_macro/f1_weighted	metrics.f1_score	F1-分数，用于多分类。多分类的 F1-分数用得较少
roc_auc	metrics.roc_auc_score	AUC 值，用得较多

### 2.3.3 回顾

- 一个良好的机器学习系统，其核心是合理的评估指标。
- 混淆矩阵：统计预测类别和真实类别的对比分布。
- 按任务目标权衡精确率和召回率。
- AUC：通过比较正负样本的排序位置，评估模型对正样本预测的可信程度。

## 2.4 回归模型

我们掌管着世界万物的运行规律，也掌管着人类的灵魂。

——电影《大鱼海棠》

前面我们专注于分类模型的介绍，在模型的训练、评估、调试三个区域介绍了很多机器学习处理数据的手段。本节开始引进回归模型，并介绍回归模型的评估指标。

### 2.4.1 回归与分类

回归问题和分类问题的区别仅仅在于设定的目标值的类型不同。分类设定的目标值是离散的，意义是“类别”；而回归设定的目标值是连续的，意义是某种“数值”。比如，对于 IRIS 花卉问题，同样的输入数据集，如果训练的目标设置成“区分花卉的种类”，那么这就是分类问题；而如果目标设置成“花朵的大小”，那么这就是一个回归问题。

由于仅仅是目标值不同，所以对于大部分机器学习模型而言，一般都有对应的“分类版”模型实现和“回归版”模型实现。例如，在 scikit-learn 封装库中，KNN 近邻模型有 KNC（K-近邻 Classification）和 KNR（K-近邻 Regression）两种对应实现。对于线性模型，同样有逻辑分类（Logistic Classification）和线性回归（Linear Regression）呼应。



相对应的分类模型和回归模型在实现思路上非常相近，一般而言，两者的主要区别在于：

- 目标数值的类型不同（离散值 vs 连续值）。
- 损失函数设计不同。
- 评估指标选择不同。

接下来以线性回归为例，介绍回归模型的应用。

## 2.4.2 线性回归

线性回归又叫作多项式回归，和逻辑分类类似，都是模型以线性函数描述数据的内在表达式，如图 2-14 所示。对比整个流程，线性回归去掉了“分数—概率”这一转换环节，输入样本数据直接经过线性函数，得到的输出数值就是模型对样本的预测值。注意，模型的预测值不再是类别的概率，而是连续数值。

$$x = [3.0, 1.0, 2.2, 4.0, \dots] \rightarrow y = wx + b \rightarrow y = 24$$

图 2-14 线性回归

简单地说，线性回归模型就是  $y = w \cdot x + b$ ，即在  $x$  的数据平面图上找一条线，尽量“拟合”所有的数据点，如图 2-15 所示。

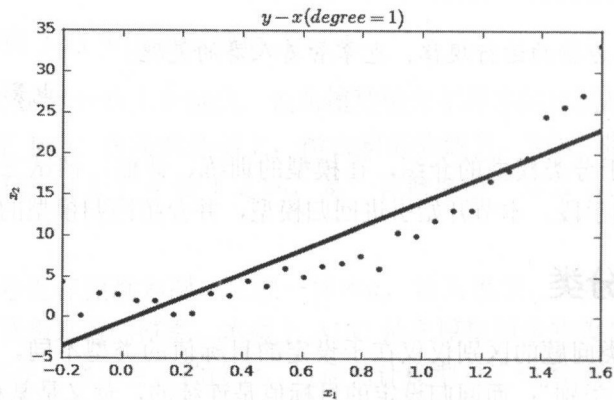


图 2-15 一次方拟合直线

前面提过，可以通过构造新特征的方式弥补线性模型对数据内部非线性关系描述不足的缺陷。因此，对于一条样本  $x = [x_1, x_2, x_3, \dots]$ ，可以构造诸如  $x_1^2$ 、 $x_2^2$ 、 $x_1x_2$  等新的多项式特征训练模型，让模型能够更好地拟合数据，如图 2-16 所示。

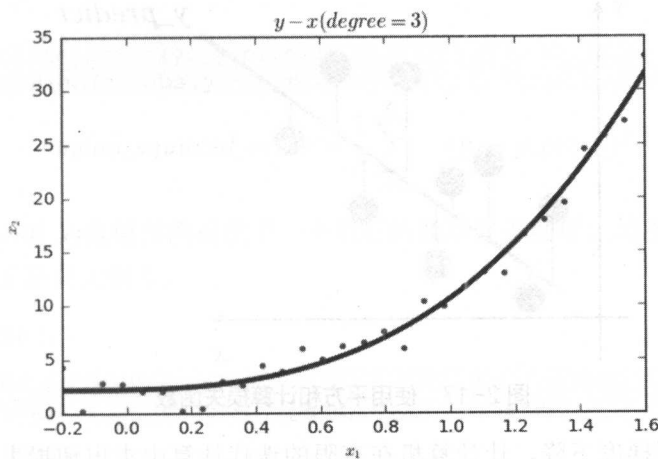


图2-16 二次方拟合

### 1. 损失函数

和分类模型一样，训练回归模型的思路是设计损失函数  $loss(w, b)$ ，通过梯度下降法寻找使  $loss(w, b)$  尽可能小的最优参数组合  $w$ 、 $b$ 。因为目标输出值不再是概率值，所以损失函数也同样会变。我们拿预测值与真实值的距离平方和来衡量模型预测值与真实值之间的差异程度。

$$loss(w, b) = -\frac{1}{M} \sum_i (y - y_{pred})^2$$

公式的代码实现：

```
import numpy as np

# X 是训练样本矩阵，w 是权重向量，b 是偏置值，y 是真实数值
def loss_func(X, w, b, y):
    y_pred = X*w+b
    return -np.mean((y_pred - y)^2)
```

为什么用平方和作为损失函数？我们可以通过直观的图形理解这一函数，衡量每个预测值与真实值之间的“距离”的平方（平方的意义在于放大那些预测偏差程度大的错误），然后将训练集上的平均距离作为预测“损失”的度量。如图2-17所示。

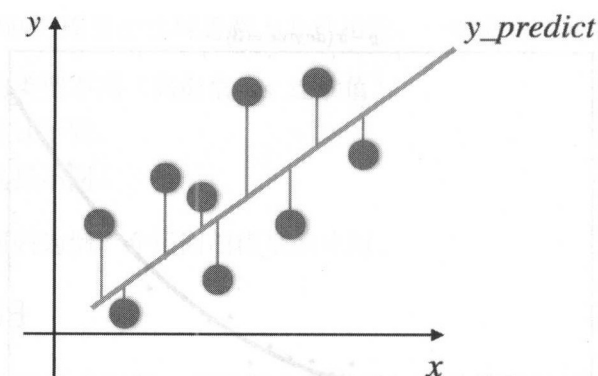


图 2-17 使用平方和计算损失函数

我们依然使用梯度下降，让计算机在有限的迭代计算中求出使损失函数尽可能小的参数组合  $w$ 、 $b$ 。

## 2. 评估指标

同样，由于目标预测值改变了，因此模型评估指标的计算方式也需要跟着改变。直接打开工具箱，常用的指标如表 2-10 所示。

表 2-10 回归评估指标

回归评估指标	scikit-learn 模块	内 容
mean_absolute_error	metrics.mean_absolute_error	误差的平均值，用得较少
median_absolute_error	metrics.median_absolute_error	误差的中位数，用得较少
mean_squared_error	metrics.mean_squared_error	均方差，常用
root_mean_squared_error	需自行实现	均方差的平方根，常用
r2	metrics.r2_score	r 方，也称确定系数，常用

具体来说，评估指标的目标是判断在测试集上，模型预测值和真实值之间的误差程度。neg\_mean\_absolute\_error 和 neg\_median\_absolute\_error 的计算方式是先算出模型预测值和真实值的绝对误差值，再取平均数/中位数：

$$\text{mean\_absolute\_error} = \frac{1}{n} \sum_{i=1}^n |y_i - y_{\text{pred}_i}|$$

平均绝对误差 (MAE)：

```
def mean_absolute_error(y, y_pred):
    return np.average(np.abs(y - y_true), axis=0)
```

$\text{median\_absolute\_error} = \text{median}(|y_i - y_{\text{pred}_i}|)$

中位绝对误差:

```
def median_absolute_error(y, y_pred):
    return np.median(np.abs(y - y_pred))
```

$$\text{mean_squared\_error} = \frac{1}{n} \sum_{i=1}^n (y_i - y_{\text{pred}_i})^2$$

MSE 和 RMSE 为数值预测提供了一个很好的通用误差度量。与 MAE 相比,它们用平方的方式放大并惩罚大误差。

均方差 (MSE):

```
def mean_squared_error(y, y_pred):
    return np.average((y_true - y_pred) ** 2, axis=0)
```

均方根差 (RMSE):

```
def root_mean_squared_error(y, y_pred):
    return np.sqrt(np.average((y_true - y_pred) ** 2, axis=0))
```

r 方 (确定系数) 是由另外两个参数 SSE 和 SST 决定的。

- SSE: 预测值和真实值的误差的平方和  $SSE = \sum_{i=1}^n (y_i - y_{\text{pred}_i})^2$ 。
- SST: 真实值和真实均值的平方和  $SST = \sum_{i=1}^n (y_i - y_{\text{mean}_i})^2$ 。

于是有:

$$r^2 = 1 - \frac{SSE}{SST}$$

其实, r 方是通过数据的变化来表征一个预测的好坏, 理论取值范围为[0 1]。需要说明的是, 由于计算机实现方式与理论计算的差异, r 方也可能是负数, 这时模型表现非常糟糕。总之, r 方越接近 1, 表明模型对数据拟合得越好。

r 方:

```
def r2_score(y, y_pred):
    sse = ((y - y_pred) ** 2).sum(axis=0, dtype=np.float64)
    sst = ((y - np.average(y, axis=0)) ** 2). \
        sum(axis=0, dtype=np.float64)
    # 特殊值处理
    if sse == 0.0:
        if sst == 0.0:
            return 1.0
        else:
            return 0.0
    return 1 - sse / sst
```

接下来，让我们看一个简单的例子，将这些知识点使用起来。

### 2.4.3 波士顿房价预测

波士顿房价数据集（Boston House Price Dataset）来源于 1978 年美国某经济学杂志上，收录在 scikit-learn 的 datasets 中。该数据集包含若干波士顿房屋的价格及其各项数据，每个数据项包含 14 个数据，分别是房屋均价及周边犯罪率、是否在河边等相关信息，其中，最后一个数据是房屋均价。

#### 第一步：描述任务

数据集，即房价相关采集数据，共 14 个特征维度，最后一维是模型将要预测的数值，即  $n=13$ 。

- CRIM：城镇人均犯罪率。
- ZN：住宅用地超过 25000 sq.ft. 的比例。
- INDUS：城镇非零售商用土地的比例。
- CHAS：查理斯河空变量（如果边界是河流，则为 1；否则为 0）。
- NOX：一氧化氮浓度。
- RM：住宅平均房间数。
- AGE：1940 年之前建成的自用房屋比例。
- DIS：到波士顿五个中心区域的加权距离。
- RAD：辐射性公路的接近指数。
- TAX：每 10000 美元的全值财产税率。
- PTRATIO：城镇师生比例。
- B：1000  $(Bk-0.63)^2$ ，其中 Bk 指代城镇中黑人的比例。
- LSTAT：人口中地位低下者的比例。
- MEDV：自住房的平均房价，以千美元计。这是模型将要预测的数值目标，即建立线性回归模型，预测波士顿房价 MEDV。

加载数据集：

```
from sklearn import datasets
from sklearn.cross_validation import train_test_split
from sklearn.preprocessing import StandardScaler
import numpy as np
import pandas as pd
from abupy import AbuML
```

```
# 数据集
scikit_boston = datasets.load_boston()
x = scikit_boston.data
y = scikit_boston.target
df = pd.DataFrame(data=np.c_[x, y],
columns=np.append(scikit_boston.feature_names, ['MEDV']))
df.head(1)
```

输出如表 2-11 所示。

表 2-11 波士顿房价数据采样

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.09	1.0	296.0	15.3	396.9	4.98	24.0

## 第二步：观察数据集

```
# 检查缺失
df.info()
```

输出：

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
CRIM      506 non-null float64
ZN        506 non-null float64
INDUS     506 non-null float64
CHAS      506 non-null float64
NOX       506 non-null float64
RM        506 non-null float64
AGE       506 non-null float64
DIS       506 non-null float64
RAD       506 non-null float64
TAX       506 non-null float64
PTRATIO   506 non-null float64
B         506 non-null float64
LSTAT     506 non-null float64
MEDV     506 non-null float64
dtypes: float64(14)
memory usage: 55.4 KB
```

## 第三步：训练模型

在模型训练过程中，执行的函数内部会使用梯度下降法寻找模型的理想参数。注意，使用梯度下降时，我们希望“数据在同一尺度上可比较”，所以这里需要归一化输入数据。

首先需要分割训练集和测试集，然后分开执行归一化不能先整体归一化数据再分割——回忆归一化数据的计算方式是使数据集整体上分布满足均值 0、方差相近的统



计属性。对于模型而言，测试集是“未知”的，当然不能把测试集的分布属性代入训练集中。

分割数据集、训练。

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2)

# 归一化数据
scaler = StandardScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.fit_transform(x_test)

# 模型训练
df = pd.DataFrame(data=np.c_[x_train, y_train],
                  columns=np.append(scikit_boston.feature_names, ['MEDV']))
boston = AbuML(x_train, y_train, df)
boston.estimator.polynomial_regression(degree=1)
reg = boston.fit()
```

#### 第四步：预测并评估模型

这里使用  $r$  方作为评估成绩。

```
# 测试集上预测
y_pred = reg.predict(x_test)

from sklearn.metrics import r2_score
r2_score(y_test, y_pred)
```

输出：

```
0.7335949812995024
```

#### 第五步：优化模型

用多项式展开的方式构造新的特征  $(x_1 + x_2 + \dots)^n = x_1^n + nx_1^{n-1}x_2 + nx_1^{n-1}x_3 \dots$ 。我们只需设置展开的多项式维度  $n$  即可。

平方展开：

```
boston.estimator.polynomial_regression(degree=2)

reg = boston.fit()
y_pred = reg.predict(x_test)
r2_score(y_test, y_pred)
```

输出：

```
0.84737704185497098
```

立方展开:

```
# r2 很差时会变成负数
boston.estimator.polynomial_regression(degree=3)

reg = boston.fit()
y_pred = reg.predict(x_test)
r2_score(y_test, y_pred)
```

输出:

```
-211.30462671802417
```

在平方展开时,模型在测试集上成绩有了不小的提升,然而在立方展开时,则明显发生了过拟合。

#### 2.4.4 泰坦尼克号生存预测:回归预测特征年龄 Age

现在,我们有了解决回归问题的能力,回到之前的“泰坦尼克号生存预测”任务中,看看之前忽略的一个问题:特征年龄 Age 的缺失。

查看特征:

```
import pandas as pd

data_train = pd.read_csv("../data/titanic/train.csv")
data_train.info()
```

输出:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
PassengerId    891 non-null int64
Survived       891 non-null int64
Pclass         891 non-null int64
Name           891 non-null object
Sex            891 non-null object
Age            714 non-null float64
SibSp          891 non-null int64
Parch          891 non-null int64
Ticket         891 non-null object
Fare           891 non-null float64
Cabin         204 non-null object
Embarked       889 non-null object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.6+ KB
```

年龄特征数据缺失约 20%, 2.4.3 节是按均值的方式补充缺失的数据,可是这种补充

方式真的适合吗？在 titanic 这种小数据量级别的数据中，填充 20% 的缺失数据，如果选择了不合适的弥补方式，那么会给模型严重的误导。可以通过一些统计图形直观衡量我们的填充方式是否破坏了数据集原有的内在属性，下面使用直方图观察原始数据年龄分布情况。

```
import seaborn as sns # Seaborn 包含了一系列的统计图形函数

sns.distplot(data_train["Age"].dropna(), kde=True, hist=True)
```

输出如图 2-18 所示。

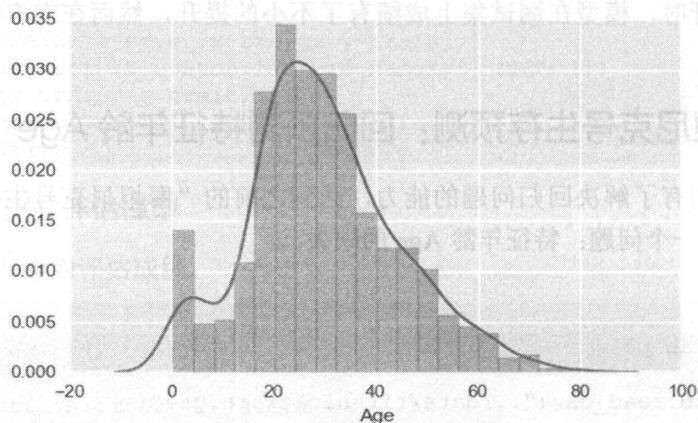


图 2-18 原始数据年龄分布

在图 2-18 中，直方图的横轴是年龄的数值，每个直方块的宽度（bins）代表一个年龄范围，如 Age: 0-5。纵轴是归一化后的数量（即该区域的样本数量除以总数），直方块就是统计样本落在每个区间的数量比例。

按均值函数填充后，观察分布：

```
def set_missing_ages(p_df):
    """均值特征填充"""
    p_df.loc[p_df.Age.isnull(), 'Age'] = \
        data_train.Age.dropna().mean()
    return p_df

data_train = set_missing_ages(data_train)
data_train_fix1 = set_missing_ages(data_train)
sns.distplot(data_train_fix1["Age"], kde=True, hist=True)
```

输出如图 2-19 所示。

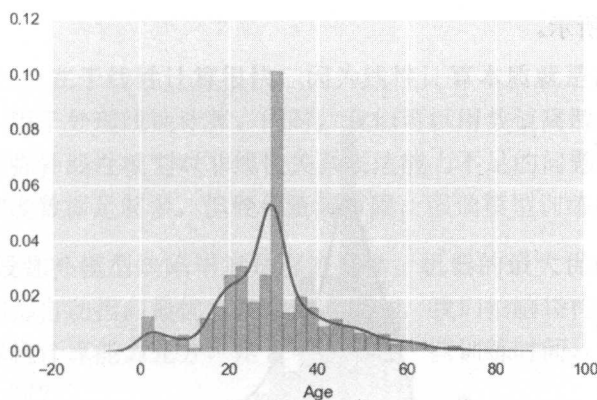


图 2-19 均值填充的年龄分布

我们无视了数据内在的联系，直接用均值补充数据，这种拍脑袋的做法显然不合理，统计的拟合曲线明显变形了。于是，下面取出一些感觉上和年龄相关的特征，比如船票、家庭成员数量等，扔进线性回归模型中，通过模型来预测年龄。

```
from abupy import AbuML
import sklearn.preprocessing as preprocessing

def set_missing_ages2(p_df):
    """回归模型预测特征填充"""
    age_df = p_df[['Age', 'Fare', 'Parch', 'SibSp', 'Pclass']]
    # 归一化
    scaler = preprocessing.StandardScaler()
    age_df['Fare_scaled'] = scaler.fit_transform(age_df['Fare'])
    del age_df['Fare']
    # 分割已有数据和待预测数据集
    known_age = age_df[age_df.Age.notnull()].as_matrix()
    unknown_age = age_df[age_df.Age.isnull()].as_matrix()
    y_inner = known_age[:, 0]
    x_inner = known_age[:, 1:]
    # 训练
    rfr_inner = AbuML(x_inner, y_inner, age_df.Age.notnull())
    rfr_inner.estimator.polynomial_regression(degree=1)
    reg_inner = rfr_inner.fit()
    # 预测
    predicted_ages = reg_inner.predict(unknown_age[:, 1::])
    p_df.loc[(p_df.Age.isnull()), 'Age'] = predicted_ages
    return p_df

data_train = pd.read_csv('../data/titanic/train.csv')
data_train_fix2 = set_missing_ages2(data_train)
sns.distplot(data_train_fix2["Age"], kde=True, hist=True)
```



输出如图 2-20 所示。

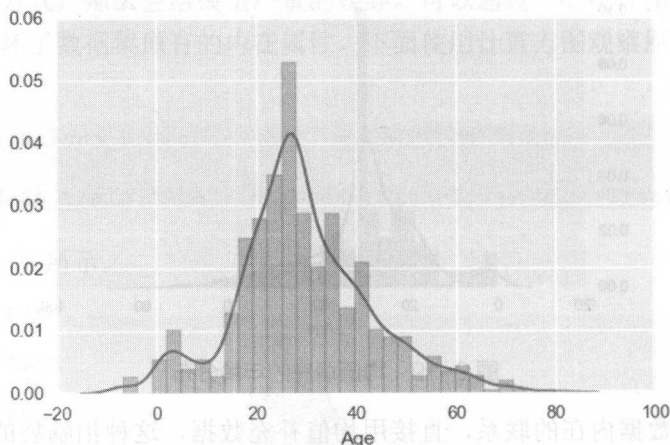


图 2-20 回归模型填充的年龄分布

是不是比直接用均值填充好了很多。下面将新的填充方式加入模型中，在相同特征情况下，对比不同的年龄处理方式对逻辑回归模型的成绩是否有影响。

如果使用 `set_missing_ages` 函数，观察均值填充的模型成绩。

```
data_train_fix1 = set_cabin_type(data_train_fix1)
train_val(data_train_fix1)
```

输出：

```
accuracy mean: 0.798073714675
```

使用 `set_missing_ages2`，观察回归模型预测填充的模型成绩。

```
data_train_fix2 = set_cabin_type(data_train_fix2)
train_val(data_train_fix2)
```

输出：

```
accuracy mean: 0.809183974577
```

是不是有了微小的进步。在经过后续几节的学习后，你还可以用一些其他的非线性模型预测回归值，使模型精益求精。

## 2.4.5 线性模型与非线性模型

至此，我们已经学习了线性的分类模型与回归模型，如何评估模型，以及一些预处理数据、构造特征等优化模型训练的技巧。现在你可以尝试将这些知识运用到你的任务中

去了。

线性模型的长处就在于线性计算很快，因为线性计算本质就是大矩阵相乘，机器有专门的硬件：显卡，用于处理线性计算。同时，我们可以用数值离散化、多项式展开等手段，人工构造新特征弥补线性模型对非线性关系表达能力不足的问题。这使得线性模型非常适用于那些特征维度数据足够多，但硬件资源有限，强调模型训练速度的使用场景。

然而，人工手段弥补模型缺陷毕竟是有迹可循。机器的最大优势就是快速地自动化，所有机器学习方向都自动指向最优、最合理。反之，我们拍脑袋构造的新特征往往很盲目。那么，如果计算机计算能力充分，是否可以牺牲一些训练时间，让模型本身有能够识别非线性关系的能力呢？

本章后续会介绍一些非线性模型。

## 2.4.6 回顾

- 分类模型与回归模型的差异。
- 目标数值的类型不同（离散值 VS 连续值）。
- 损失函数设计不同。
- 评估指标设计不同。
- 线性回归模型。
- 常用的回归模型评估指标：均方差、 $r$ 方。
- 可以用统计图形直观估计数据处理手段的影响。

## 2.5 决策树模型

一个成功的决策，等于 90% 的信息加上 10% 的直觉。

——美国企业家 S·M·沃尔森

本节我们将接触第一个参数化的非线性模型。

非线性模型将线性模型需要构造非线性特征的负担扛了过来，代价是训练速度变慢。非线性模型的构造不同于以往的多项式的表达方式，我们需要从一个新的视角，描述事物的内在规律。



## 2.5.1 信息与编码

给定一个标定类别的数据集 $X$ ，如何计算出数据集中信息量的大小呢？前辈香农给出了一套基于概率计算信息量长度的公式：

$$H(X) = - \sum_i P(x_i) \log_2 P(x_i) \quad (P(x_i): \text{事件 } x_i \text{ 出现的概率})$$

信息熵函数：

```
import numpy as np

def entropy(P):
    """根据每个样本出现的概率，计算信息量，
    输入的 P 是数据集上每个数值统计的频率（概率）向量"""
    return -np.sum(P * np.log2(P))
```

让我们通过一个小故事理解上面的公式。

考场上，卷面有 100 道考题，每个题目有 ABCD 四个选项，出题老师提前说明每个选项都可能出现。我在给死党小明做小抄传答案，希望小抄尽可能短，不容易被监考老师抓到。而我其实是一个隐藏在人类世界的机器人，不会写字符语言如“A、B、C、D”，只会写 0、1 两种数字（用二进制位编码事件）。那么我应该如何传递答案，才能让小抄最短呢？

如果依然让我传递 ABCD 的 ASCII 码的话，每个答案需要占 8 位，如“01100001”表示“A”。从传输的角度，这显然很浪费，一些如“E、F”等后续的字母根本用不上。信息学最初要解决的问题，就是数据的压缩和传输。编码 4 个答案选项其实只需用 2bit（位）就可以了：00-A,01-B,10-C,11-D。

我传给小明的小抄内容：10110001.....1011（CDAB.....CD）。

用香农公式计算一下这个小抄的信息量：由于每个选项都可能出现，所以 $P(A) = P(B) = P(C) = p(D) = \frac{1}{4}$ ，小抄的信息量是 $H(X) = -\sum_{i=1}^4 \frac{1}{4} \cdot \log_2(\frac{1}{4}) = 2$ 。

```
p = [0.25, 0.25, 0.25, 0.25] # ABCD 的出现概率
H = entropy(p)
print '小抄的信息量: {}'.format(H)
```

输出：

```
小抄的信息量: 2.0
```

100 道题的答案就是 100 个样本的标签。所以这段故事隐含了一个概念：若数据集中，

样本标签的信息量的多少=编码数据集中所有样本标签所需要的最短字符的长度。

上面的例子中，100 道题目的答案中有多少信息量=最少需要用几 bit 才能描述清楚所有题目的答案。也就是说，信息量是一个可以度量的编码长度，以 bit 为单位。

我在考前研究了老师的“出题特征”，发现奇数项考题的答案不是“A”就是“C”，偶数项考题的答案不是“B”就是“D”。对这一特征进行分析之后，我决定缩短小抄的编码方式为：0-A 或 B，1-C 或 D。小抄的内容如：1100.....11 (CDAB.....CD)。小明拿到小抄，对着“题目编号是奇数还是偶数”这一特征，将小抄答案解读出来。

注意，当我分析出题目集的一个有效“特征”之后，考场上传递给小明的小抄的编码长度立马缩减了。特征“奇数项考题 AC，偶数项考题 BD”将原来的套题集划分成奇数集和偶数集两块，如图 2-21 所示。通过香农公式计算下整体的信息量：

$$\begin{aligned}
 H'(X) &= \sum H_i(X) = \frac{1}{2} \cdot H(\text{奇数题集}) + \frac{1}{2} \cdot H(\text{偶数题集}) \\
 &= \frac{1}{2} \cdot \left( - \sum_{i=1}^2 \frac{1}{2} \cdot \log_2 \left( \frac{1}{2} \right) \right) + \frac{1}{2} \cdot \left( - \sum_{i=1}^2 \frac{1}{2} \cdot \log_2 \left( \frac{1}{2} \right) \right) = 1
 \end{aligned}$$

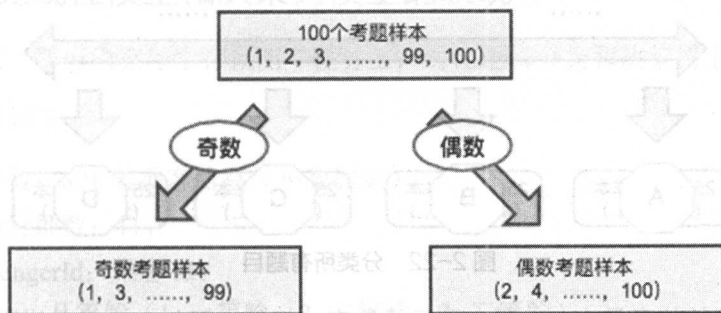


图 2-21 按奇偶特征分类题目

```

p_1 = [0.5, 0.5] # AC 的出现概率
frac_1 = 50.0 / 100.0 # 奇数题集的比例

```

```

p_2 = [0.5, 0.5] # BD 的出现概率
frac_2 = 50.0 / 100.0 # 偶数题集的比例

```

```

H = frac_1 * entropy(p_1) + frac_2 * entropy(p_2)
print '小抄的信息量: {}'.format(H)

```

输出：

```
小抄的信息量: 1.0
```

也就是说，按有效的特征条件划分数数据集时，数据集的标签信息量会减少。这个特征越有效，信息量减少的越多。

我和小明一起研究了出题老师的一些其他特征，比如“如果奇数题答案是 A，偶数题目答案就一定是 B；如果奇数题答案是 C，偶数题答案就一定是 D”，将编码长度进一步缩短为 0.5 (0-AB, 1-CD, 小抄内容为 10.....1-CDAB.....CD)，小抄的信息量进一步下降。渐渐地，当分析的特征足够多后，小明发现他已经不需要我传的小抄，就知道所有题目的答案了。如图 2-22 所示。

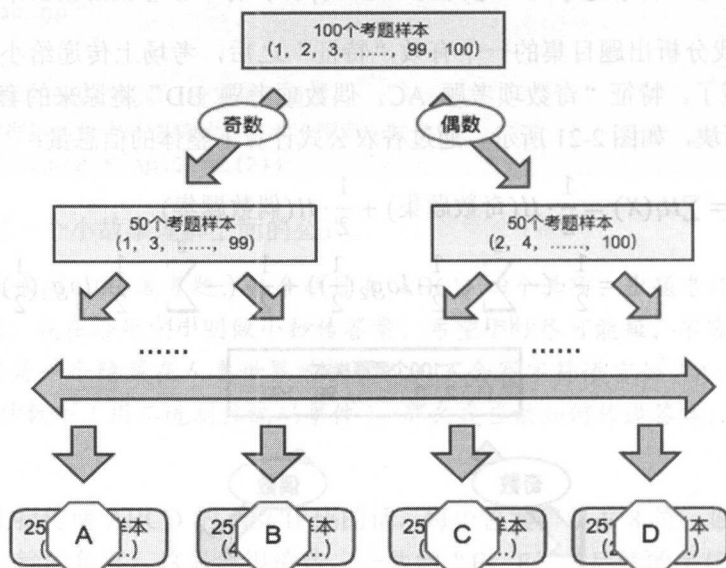


图 2-22 分类所有题目

这时，小抄的信息量为 0，所有题目的答案（标签）已知。也就是说，对于一个样本集，分类出所有样本的标签就是为了让样本集标签相关的信息量缩减至 0。本质上，信息是描述“不确定性”的一个概念，所以当所有样本标签都已知时，对应的信息量也就是 0 了。

举例传小抄只是为了有趣，做人应该诚信唯美。

## 2.5.2 决策树

上面的故事是通过特征分割样本集，缩减样本集的信息量，最终分类出所有样本的一个例子。决策树模型正是按照这一思路设计的，即寻找样本特征中使整体样本集信息下降最快的特征作为树的节点。决策树的算法执行流程如下。

- (1) 起始  $N=1$  个样本集群  $S(N=1)$  和一个特征集  $F\{f_1, f_2, \dots\}$ 。
- (2) 对于  $N$  个样本集，依次计算每个特征作为分割节点，使整体样本下降的信息量。
- (3) 选择一个使决策树信息量下降最快的特征  $f_i$  和样本集进行分割。
- (4) 从特征集中去掉  $f_i$ ，去掉旧样本集，新生成的两个样本子集加入  $S(N) \rightarrow S(N+2-1=N-1)$ 。
- (5) 递归执行 (2) 至 (4) 步，直至特征集为空或者到达指定的树深度。

对于工程师而言，其实只要理解决策树是不断选择使数据集整体信息量下降最快的特征作为节点建立的树模型，这一核心点就足够了。在工程中，直接调用决策树模型的封装类就可以使用。决策树和大部分机器学习模型一样，可以用于分类和回归问题上，对应的实现封装在 `sklearn.tree.DecisionTreeClassifier` 模块和 `sklearn.tree.DecisionTreeRegressor` 模块中。

### 2.5.3 对比线性模型和决策树模型的表现

下面将在“泰坦尼克号生存预测”任务上，对比逻辑分类和决策树上的表现成绩。

任务描述如下。

- 样本数：891 名乘客信息（小数据集）。
- 原始特征数：11。
- PassengerId：乘客 id。
- Pclass：几等舱（1—一等舱，2—二等舱，3—三等舱）
- Name：名字。
- Sex：性别。
- Age：年龄。
- SibSp：兄弟姐妹/配偶的数量。
- Parch：父母/孩子的数量。
- Ticket：机票号码。
- Fare：票价。
- Cabin：客舱。
- Embarked：登船的港口（C = 瑟堡，Q = 皇后镇，S = 南安普顿）。
- 目标：预测 Survived（1—生存，0—死亡）。

在同样的特征集下，我们先看线性模型基于交叉验证（Cross-validation）（参见 2.2 节）的准确率。

```
titanic.estimator.logistic_regression()
titanic.cross_val_accuracy_score()
```

输出：

```
accuracy mean: 0.809183974577
array([0.83333333, 0.81111111, 0.78651685, 0.84269663, 0.82022472,
       0.7752809, 0.78651685, 0.80898876, 0.80898876, 0.81818182])
```

接着来看非线性模型：决策树的表现成绩、scikit-learn 封装的决策树参数 criterion。选择决策树衡量最佳分裂特征的方式：gini（默认，基尼杂质）或者 entropy（信息减少量）。gini 是从另一个角度评估选择最适合的特征，两者效果差异不大，这里选择 entropy。

注意，这里用 grid search 搜索决策树的最优层数深度参数 max\_depth。

切换决策树：

```
titanic.estimator.decision_tree_classifier(criterion='entropy')
# grid search 寻找最优的决策树层数
param_grid = dict(max_depth=range(3, 10))
best_score_, best_params_ = titanic.grid_search_common_clf(
    param_grid, cv=10, scoring='accuracy')
best_score_, best_params_
```

输出：

```
(0.81144781144781142, {'max_depth': 3})
```

输入：

```
titanic.estimator.decision_tree_classifier(criterion='entropy',
                                           **best_params_)
titanic.cross_val_accuracy_score()
```

输出：

```
accuracy mean: 0.811443366247
array([0.81111111, 0.81111111, 0.78651685, 0.85393258, 0.82022472,
       0.79775281, 0.79775281, 0.78651685, 0.84269663, 0.80681818])
```

可以看到，在 titanic 数据集上，相同特征条件下，非线性模型确实比线性模型表现得更好一点。我们可以画出决策树的逻辑，看看模型具体如何选择特征层级。

画出决策树：



```
# 依赖 Python 的 pydot 和 graphviz 包
from sklearn import tree
import pydot
from sklearn.externals.six import StringIO

# 为了方便, 这里限制决策树的深度观察
titanic.estimator.decision_tree_classifier(criterion='entropy', max_depth=3)
clf = titanic.fit()

# 存储树 plot
dotfile = StringIO()
tree.export_graphviz(clf, out_file=dotfile,
feature_names=titanic.df.columns[1:])
pydot.graph_from_dot_data(dotfile.getvalue()).write_png("dtree2.png")
!open dtree2.png
```

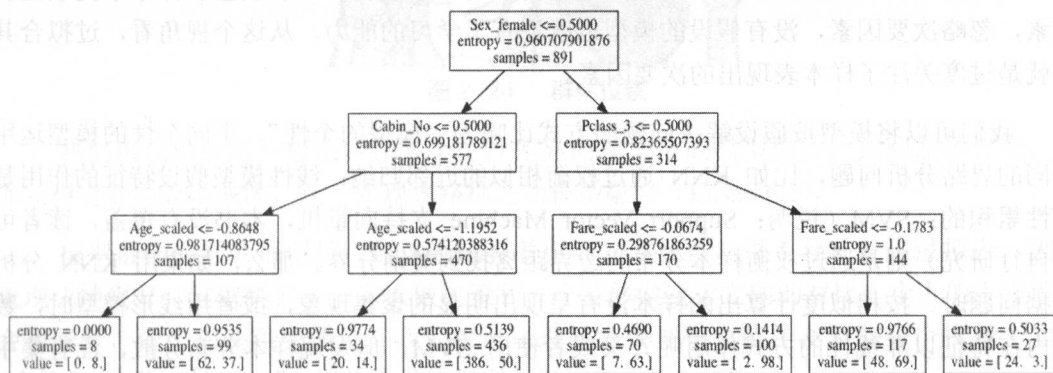


图 2-23 titanic 决策树图

可以看到, 模型先选择“是否是女性”作为第一特征, 后来依次选择“是不是三等舱”等其他特征分割数据集。每层整体数据集的信息量 (entropy), 即每层每个数据集信息量的概率和在逐渐减少, 最终构建出决策树模型。

在这个任务中, 非线性模型对比线性模型只有了微弱的提升。在实际工程中其实也是如此, 原始的数据总量、数据质量、特征质量很大程度上决定了模型的表现成绩。一个采集了海量数据和高维度的优质特征的数据集下, 简单的模型就能表现得很好了, 复杂的模型只是在百尺竿头更进一步。

## 2.5.4 回顾

- 信息量是一个可以度量的编码长度, 以 bit 为单位。
- 对于一个样本集, 分类出所有样本的标签就是让标签数据集的信息量缩减至 0。
- 决策树通过不断选择使数据集整体标签信息量下降最快的特征作为节点建树。



## 2.6 模型融合

三个臭皮匠，顶个诸葛亮。

——谚语

机器学习中还有很多其他模型，读者完全可以举一反三，自行学习下去，本书篇幅有限，不再一一介绍。本节介绍融合模型的技巧，由于这些技术手段（英文的）中文释义在笔者看来很不合适，因此下面会用英文原文介绍。

模型学习的本质和人相似，就是“归纳”。而归纳是需要假设前提的，比如假设新的事物的分布符合某种规律。每个模型都有自己的归纳假设，帮助模型从样本中提取主要因素，忽略次要因素，没有假设的模型也就没有了学习的能力。从这个视角看，过拟合其实就是过度关注了样本表现出的次要因素。

我们可以将模型按假设解决问题的方式比喻为“模型的个性”。不同个性的模型运用不同的思路分析问题，比如 KNN 通过权衡相似的近邻归纳；线性模型假设特征的作用是线性累积的；SVM（说明：Support Vector Machine 支持向量机，本书没有覆盖，读者可以自行研究）则是通过权衡样本分布的边界距离找到类别分界。那么，如果用 KNN 分析数据问题时，按相似度计算出的样本没有呈现出明显的聚集现象，或者用线性模型时，数据内在特征以非线性的方式作用呢？又或者使用 SVM，但发现样本分布分散，没有集中在数据的边界呢？

模型的个性带来的问题是模型只会以自己擅长的模式解读样本，它们并不像人的思维那么辩证。于是显而易见的缺陷就是，当数据分布适应模型的个性时，成绩理想；反之，模型成绩变得糟糕。

很多时候，面对样本分布倾向并不明确的任务，我们需要消除模型的独立个性，让模型学会用多个视角观察样本集。

### 2.6.1 融合成群体 (Ensamble)

数学中有个简单有趣的理论，叫作“孔多塞陪审团定理”。大意是说：如果一个群体中每个人做出正确判断的概率高于 50%，成为“大概率正确”，那么这个群体通过投票会议获得的正确率就会融合多个大概率。如果群体中的个体无限多，那么群体的正确率可以达到 100%。因为群体中每个人决策判断过程中，错误的部分是有个体差异的，并且比例总是少数的，因而群体的个体之间投票时，会用多数的正确票遮盖少数的错误票，最终群

体表现可以获得高于个体表现的成绩。

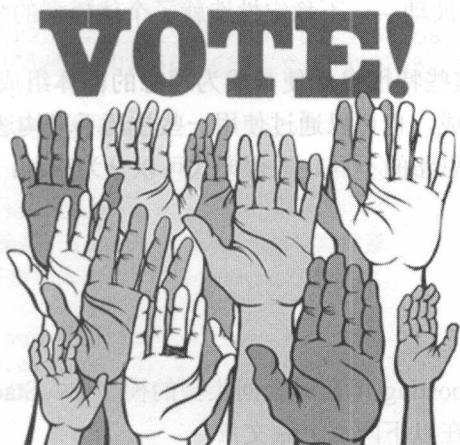


图 2-24 群体投票

我们可以借鉴这一想法，让多个个性差异的模型形成模型群体，共同发挥作用，从而获得更好的表现成绩，这一实现思路称之为模型融合（Ensamble）。

模型群体的好处很明显，群体系统有着更高的健壮性，更不容易发生过拟合，因此表现成绩更好。这就类似于一个人做决策和一个群体做决策之间的对比，个人的主观偏见会被降低，做出的决策更通用。我们也可以理解为模型融合就是在消除每个模型个体的“个性”，去除“过拟合”，使整体最终表现出更好的成绩。

简单梳理下，模型的个性来源是个体模型之间分析处理数据集的方式和角度的差异，所以群体对个体模型的要求是：

- 个体正确率大于 50%。
- 个体判断问题存在差异，有“个性”。

这种尤其适合非线性模型，因为模型的表达能力非常强，更容易过度拟合数据集，具体表现为成绩在不同数据集中波动程度更大，适应力更低。

举个例子：决策树模型。决策树可以通过构造极度丰富的特征和极深的树深度使其训练集成绩接近满分；但对应的模型适应力最低。通常情况下，决策树靠上的特征层级晃动时，也会造成成绩的巨大差异。比如在 2.5.3 节“泰坦尼克号生存预测”问题中，如果不把“是否为女性”作为顶层特征节点，那么成绩差异就会非常大。简单地说，类似决策树这样的非线性模型的特点是：

- 模型表达能力很强，但很容易过拟合数据。
- 模型成绩很容易波动——不稳定性造就了个体模型的个性。

能力强、有个性，这些特性自然使其成为理想的群体组成成员。模型群体即可以通过不同类型的模型个体组成，也可以通过使用一些技巧手段构造出的有差异的同类模型组成。具体实现起来，按照思路的不同，融合方式可以分为三种：

- Bagging
- Boosting
- Stacking

其中，Bagging 和 Boosting 用在融合同类型的模型中，Stacking 用在融合不同类型的模型中。这些手段基本都在以下两点上做文章：

- (1) 如何保证个体差异。
- (2) 以什么方式融合差异个体的投票，作为最终判断。

## 2.6.2 Bagging：随机森林（Random Forest）

Bagging 应用于同类型的模型个体，使其形成模型群体。在 Bagging 中，每个模型不再训练全部的训练集样本，而是在训练集的有放回抽样的随机子集中训练模型；预测样本时等权重投票。也就是说，Bagging 通过构造数据集的随机子集保证模型个体差异，等权重投票融合模型个体的预测。

基于 Bagging 思路改进实现的决策树群体模型就是随机森林（Random Forest）。随机森林在实现上不仅使用训练集的子集，同时在决策树建立树节点时只在特征集的子集中挑选。由于决策树特征层级（特征节点的选择顺序）的特性是对成绩影响很大，因此这种方式使构造出的个体内在的个性很鲜明，是理想的群体成员。下面仍然以泰坦尼克号生存预测为例，观察其表现。

先看下决策树的成绩：

```
# 决策树
titanic.estimator.decision_tree_classifier()
# grid search 寻找最优的决策树层数
param_grid = dict(max_depth=range(3, 10))
_, best_params_ = titanic.grid_search_common_clf(param_grid, cv=10,
                                                  scoring='accuracy')
titanic.estimator.decision_tree_classifier(**best_params_)
titanic.cross_val_accuracy_score()
```

输出:

```
accuracy mean: 0.812580013619
array([0.81111111, 0.82222222, 0.7752809, 0.85393258, 0.82022472,
       0.7752809, 0.79775281, 0.78651685, 0.85393258, 0.82954545])
```

同样特征条件下, 切换随机森林:

```
# 随机森林
titanic.estimator.random_forest_classifier()
# grid seach 寻找最优的参数: n_estimators 个体模型数量
# max_features 特征子集样本比例; max_depth 层数深度
param_grid = {
    'n_estimators': range(80, 150, 10),
    'max_features': np.arange(.5, 1., .1).tolist(),
    'max_depth': range(1, 10) + [None]
}
# n_jobs=-1 开启多线程
best_score_, best_params_ = \
    titanic.grid_search_common_clf(param_grid, cv=10,
                                   scoring='accuracy', n_jobs=-1)
best_score_, best_params_
```

输出:

```
(0.83501683501683499,
 {'max_depth': 8, 'max_features': 0.6, 'n_estimators': 80})
```

随机森林的表现成绩:

```
titanic.estimator.random_forest_classifier(**best_params_)
titanic.cross_val_accuracy_score()
```

输出:

```
accuracy mean: 0.830657984338
array([0.78888889, 0.77777778, 0.75280899, 0.85393258, 0.93258427,
       0.83146067, 0.82022472, 0.79775281, 0.8988764, 0.85227273])
```

与之前的成绩相比, 我们又有了一小步的提升, 这是群体智慧的胜利!

## 2.6.3 Boosting: GBDT

Bagging 利用随机采样集合的方式来构造个体差异, 一些反对者认为这种随机性虽然保证了“个性”, 却显得盲目。Boosting 的思路很有针对性: 依次生成一个模型个体序列  $M(M_1, M_2, \dots, M_n)$ , 其中, 后续模型会尝试修正前面模型的错误。

Boosting 有很多种版本实现, 最常用的是 Ada-Boosting 和 Gradient-Boosting, 可以

用于分类问题和回归问题。接下来以分类问题为例，直观地阐述这些实现的执行流程。

假设数据集样本总数为  $m$ ，要构造的模型群体中模型个数为  $n$ 。

目标：获得融合后的模型  $M(M_1, M_2, \dots, M_n)$ 。

## 1. Ada-Boosting

Ada-Boosting 的核心思想是使序列的下一个模型更关注之前的错误样本。具体的执行步骤如下。

(1) 为整个数据集的每个样本分配一个权重  $s_1, s_2, \dots, s_m$ ，直观理解：权重代表模型在训练时对样本的“重视度”。

(2) 在整个数据集上训练模型  $m_i$ ，并执行分类预测，对应的正确率为  $acc_i$ 。

(3) 评估模型  $M_i$  对每个样本  $j$  的分类正确与否，如果分错该样本，则增大样本对应的权重  $s_j$ ，即让下一个模型更关注之前的错误样本。

(4) 重复步骤 2 至 3，直至构造出  $n$  个模型个体。

(5) 归一化序列  $[acc_1, acc_2, \dots, acc_n]$ ，使其累加和等于 1，得到新的序列  $[w_1, w_2, \dots, w_n]$ ，最终模型对某个类别的预测概率值等于每个模型预测的加权求和： $M(M_1, M_2, \dots, M_n) = \sum_i w_i \cdot M_i$

对于回归问题，替换第 2 步的评估指标，最终模型按每个模型个体预测值加权求和得出即可。从上面的流程可以想象得到，因为每个模型对样本的关注权重不同，所以保证了模型个体的差异。同时，这个执行流程越往后执行，训练出的模型就越会在意那些前面模型容易分错（权重高）的样本。

## 2. Gradient-Boosting

Gradient-Boosting 直观的执行流程更为简洁（数学推导及实现却更为复杂），可以直接用于概率分类或者回归问题，其核心思想是：序列的后续模型不再直接预测数据集的预测值，而是预测之前模型的预测值和真实值的差值。

训练模型  $M_1$ ，得出预测值，这时模型的整体预测值为： $y_{pred} = y_{pred_1}$ 。

计算上一步预测值和真实值的差值： $dy_1 = y_{pred_1} - y$ ，将这个值作为模型  $M_2$  待预测的目标值，模型转而开始预测之前模型的预测值和真实值的差值。这时，总模型的预测值为  $y_{pred} = y_{pred_1} + dy_{pred_2}$ 。



模型  $M_{i+1}$  预测前  $i$  个模型预测值之和与真实值之间的差值，这一步用到了梯度下降和模型前  $i$  个模型中预测值和真实值的差值：  $dy_1, \dots, dy_i$ ，获得预测值  $dy\_pred_{i+1}$ 。

重复第二步，得到多个预测数值序列  $[dy_1, dy_2, \dots, dy_n]$ ，最终的融合预测为：  
 $y\_pred = y\_pred_1 + \sum_i dy\_pred_i$ 。

将 Gradient-Boosting 用于决策树模型上之后，就是 GBDT (Gradient Boosting Decision Tree 梯度增强决策树)。接下来看看这个模型在 titanic 任务上的表现成绩。

切换 GBDT:

```
# GBDT
titanic.estimator.gbdt_classifier()

# grid search 寻找最优的参数: n_estimators 个体模型数量; max_depth 层数深度
param_grid = {
    'n_estimators': range(80, 150, 10),
    'max_depth': range(1, 10)
}

# n_jobs=-1 开启多线程, cv 缩小到 5, for speed
best_score_, best_params_ = \
    titanic.grid_search_common_clf(param_grid, cv=5,
                                   scoring='accuracy', n_jobs=-1)

best_score_, best_params_
```

输出:

```
(0.83501683501683499, {'max_depth': 5, 'n_estimators': 140})
```

查看训练成绩:

```
titanic.estimator.gbdt_classifier(**best_params_)
titanic.cross_val_accuracy_score()
```

输出:

```
accuracy mean: 0.828361139485
array([0.82222222, 0.8, 0.76404494, 0.84269663, 0.8988764,
       0.82022472, 0.83146067, 0.78651685, 0.85393258, 0.86363636])
```

按照经验来看，大部分分类任务中，GBDT 的成绩和随机森林差别不大，仅在有些任务中 GBDT 能够略好一点。

实际使用中大家可以感觉到，由于 boosting 的模型序列有前后依赖，不好实现并发，因此在开启多线程下同样级别参数的 GridSearch 中，GBDT 比随机森林明显要慢得多。



在上面的例子中，`gbdt_classifier` 函数使用了一个 `sklearn.ensemble` 包之外的 GBDT 实现类库 XGBoost。XGBoost 是目前非常流行的机器学习库，它对并行计算做了专门的优化，并且在 `xgboost.sklearn` 中有完全 `scikit-learn` 风格的封装接口。

## 2.6.4 Stacking

Stacking 用于不同类型的模型个体之间的融合。

前面几节我们接触了线性模型、决策树、随机森林、GBDT 等模型，这些模型对数据集都有着不同类型的“先天偏见”。比如线性类模型总是认为数据集的内在表达式是线性的、决策树系的模型则认为内在属性可以通过树形表示——对事物的特定解读方式本身就是带着一种特定偏见。不同类型的模型融合的意义在于通过群体形成的智慧弥补个体自身的先天缺陷。

Stacking 实现起来很简单，模型的先天偏见保证了成员之间的个性差异。将不同模型训练的结果以某种方式融合在一起就可以了，常用的融合方式有两种。

(1) 加权投票：将每个模型的成绩作为权重，最终预测值是每个模型乘以权重，然后相加。

(2) 通过一个新的模型融合多个模型个体，新模型一般是线性模型。

第二种方法一般是用线性模型（逻辑分类/线性回归），将每个模型对样本的预测作为特征输入，真实值作为待预测的  $y$ 。用简单线性模型的原因是模型内在可挖掘的非线性因素已经被模型群体充分挖掘了，剩下的用线性模型对每个成员合理分配权重就可以了。

下面将在 `titanic` 任务上实现第二种融合方法，第一种融合方式可以参见 `sklearn.ensemble.VotingClassifier` 模块，实际中一般也是第二种融合方式效果更好一些。

首先，选择下面几个分类器。

逻辑分类：

```
titanic.estimator.logistic_regression()
titanic.cross_val_accuracy_score()
```

输出：

```
accuracy mean: 0.806974804222
array([0.82222222, 0.81111111, 0.7752809, 0.83146067, 0.80898876,
       0.76404494, 0.78651685, 0.79775281, 0.83146067, 0.84090909])
```

随机森林：

```
param = {'max_depth': 8, 'max_features': 0.6, 'n_estimators': 80}
titanic.estimator.random_forest_classifier(**param)
titanic.cross_val_accuracy_score()
```

输出:

```
accuracy mean: 0.828411077063
array([0.8, 0.77777778, 0.74157303, 0.86516854, 0.91011236,
       0.84269663, 0.80898876, 0.79775281, 0.87640449, 0.86363636])
```

GBDT:

```
param = {'max_depth': 5, 'n_estimators': 140}
titanic.estimator.gbdt_classifier(**param)
titanic.cross_val_accuracy_score()
```

输出:

```
accuracy mean: 0.82947196686
array([0.8, 0.82222222, 0.76404494, 0.84269663, 0.8988764,
       0.80898876, 0.82022472, 0.82022472, 0.86516854, 0.85227273])
```

将这几个模型的预测值作为  $x$ 。注意, 这里要融合的是模型预测的概率值, 而不是标签值 (可以认为标签值丢失了一部分信息), 真实标签依然是  $y$ , 做逻辑分类。

# 准备训练好的模型

```
titanic.estimator.logistic_regression()
lr = titanic.fit()
param = {'max_depth': 8, 'max_features': 0.6, 'n_estimators': 80}
titanic.estimator.random_forest_classifier(**param)
rf = titanic.fit()
param = {'max_depth': 5, 'n_estimators': 140}
titanic.estimator.gbdt_classifier(**param)
gbdt = titanic.fit()
```

# 构造 stacking 训练集, 将融合三个模型预测的概率值作为特征数据

```
x_stk = np.array(
    [lr.predict_proba(x)[:, 0], rf.predict_proba(x)[:, 0],
     gbdt.predict_proba(x)[:, 0]]).T
x_df_stk = pd.DataFrame(x_stk, columns=['lr', 'rf', 'gbdt'])
y_df = pd.DataFrame(y, columns=['y'])
df = y_df.join(x_df_stk)
```

# stacking 模型

```
stackings = AbuML(x_stk, y, df)
stackings.estimator.logistic_regression()
```

# 获得 titanic 的融合模型 stk

```
stk = stackings.fit()
```

stk 就是我们在已有数据集上能够得到的最好模型。

### 一个常见的错误

有了融合后的模型后，我们希望对比一下单个模型的表现成绩，这时需要在新的数据集而不是在原有的数据上做 cross\_validation 分数测试，为什么呢？

查看 stacking 成绩：

```
stackings.cross_val_accuracy_score()
```

输出：

```
accuracy mean: 0.94500453978
array([0.92222222, 0.95555556, 0.91011236, 0.94382022, 0.97752809,
       0.96629213, 0.93258427, 0.93258427, 0.97752809, 0.93181818])
```

多么惊人的提升。但这个提升合理么？与之前的模型不同，stk 模型的输入特征是在整体数据集上训练出来的模型的预测值。也就是说，stk 的每个输入样本的数据都是在整个数据集上训练的成果，每条样本都隐含了整个数据集的信息。在这上面做训练-测试划分显然是没有意义的，不管什么样的划分方式，都达不到我们期望的训练集与测试集信息完全隔离的效果，这种导致成绩虚高的现象叫作“信息泄露”。

解决方法如下，通过 K-folder 分割数据集，在封闭的训练集上完成模型的训练及融合，然后在测试集上对比成绩。

```
from sklearn.cross_validation import KFold
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn import metrics
from sklearn.grid_search import GridSearchCV

def lr_model(x_train, x_test, y_train, y_test):
    """返回训练好的逻辑分类模型及分数"""
    lr = LogisticRegression(C=1.0)
    lr.fit(x_train, y_train)
    y_pred = lr.predict(x_test)
    return lr, metrics.accuracy_score(y_test, y_pred)

def rf_model(x_train, x_test, y_train, y_test):
    """返回训练好的随机森林模型及分数"""
    param_grid = {
        'n_estimators': range(80, 120, 10),
        'max_features': np.arange(.6, .9, .1).tolist(),
        'max_depth': range(3, 9) + [None]
    }
```

```

grid = GridSearchCV(RandomForestClassifier(), param_grid, cv=10,
                    scoring='accuracy', n_jobs=-1)
grid.fit(x_train, y_train)
rf = RandomForestClassifier(**grid.best_params_)
rf.fit(x_train, y_train)
y_pred = rf.predict(x_test)
return rf, metrics.accuracy_score(y_test, y_pred)

def gbd_t_model(x_train, x_test, y_train, y_test):
    """返回训练好的GBDT模型及分数"""
    param_grid = {
        'n_estimators': range(80, 120, 10),
        'max_features': np.arange(.6, .9, .1).tolist(),
        'max_depth': range(3, 9) + [None]
    }

    grid = GridSearchCV(GradientBoostingClassifier(), param_grid,
                        cv=10, scoring='accuracy', n_jobs=-1)
    grid.fit(x_train, y_train)
    gbd_t = GradientBoostingClassifier(**grid.best_params_)
    gbd_t.fit(x_train, y_train)
    y_pred = gbd_t.predict(x_test)
    return gbd_t, metrics.accuracy_score(y_test, y_pred)

def stack_models(x_train, x_test, y_train, y_test):
    """返回融合后的模型及分数"""
    param_grid = {
        'C': [.01, .1, 1, 10]
    }

    grid = GridSearchCV(LogisticRegression(), param_grid, cv=10,
                        scoring='accuracy', n_jobs=-1)
    grid.fit(x_train, y_train)
    # L1 正则化, 更稀疏
    stk = LogisticRegression(penalty='l1', tol=1e-6,
                             **grid.best_params_)
    stk.fit(x_train, y_train)
    y_pred = stk.predict(x_test)
    return rf, metrics.accuracy_score(y_test, y_pred)

kf = KFold(len(titanic.y), n_splits=5, shuffle=True)
lr_scores = []
rf_scores = []
gbd_t_scores = []
stk_scores = []

for train_index, test_index in kf:
    x_train, x_test = titanic.x[train_index], titanic.x[test_index]
    y_train, y_test = titanic.y[train_index], titanic.y[test_index]

```



```

# 单个模型成绩
lr, lr_score = lr_model(x_train, x_test, y_train, y_test)
rf, rf_score = rf_model(x_train, x_test, y_train, y_test)
gbdt, gbdt_score = gbdt_model(x_train, x_test, y_train, y_test)

# stacking
x_train_stk = np.array([lr.predict_proba(x_train)[: , 0],
                        rf.predict_proba(x_train)[: , 0],
                        gbdt.predict_proba(x_train)[: , 0]]).T
x_test_stk = np.array([lr.predict_proba(x_test)[: , 0],
                      rf.predict_proba(x_test)[: , 0],
                      gbdt.predict_proba(x_test)[: , 0]]).T
stk, stk_score = stack_models(x_train_stk, x_test_stk, y_train,
                             y_test)

# append score
lr_scores.append(lr_score)
rf_scores.append(rf_score)
gbdt_scores.append(gbdt_score)
stk_scores.append(stk_score)

print 'lr mean score: {}'.format(np.mean(lr_scores))
print 'rf mean score: {}'.format(np.mean(rf_scores))
print 'gbdt mean score: {}'.format(np.mean(gbdt_scores))
print 'stk mean score: {}'.format(np.mean(stk_scores))

```

输出:

```

lr mean score: 0.803540267403
rf mean score: 0.809177076141
gbdt mean score: 0.813684012303
stk mean score: 0.801330738811

```

在这个数据任务中，stacking 模型融合对准确率成绩似乎没有多大影响，下面比较一下成绩在不同 folder 的具体波动程度：

```

print 'lr std score: {}'.format(np.std(lr_scores))
print 'rf std score: {}'.format(np.std(rf_scores))
print 'gbdt std score: {}'.format(np.std(gbdt_scores))
print 'stk std score: {}'.format(np.std(stk_scores))

```

输出:

```

lr std score: 0.0482251333467
rf std score: 0.0319151813243
gbdt std score: 0.0223669765661
stk std score: 0.0170609211563

```

可以看到，融合后的模型方差更小，成绩更稳定。下面展开几个模型的成绩观察下。

随机森林的各个 folder 成绩：

```
rf_scores
```

输出:

```
[0.83240223463687146,
0.84269662921348309,
0.797752808988764,
0.8202247191011236,
0.7528089887640449]
```

GBDT 的各个 folder 成绩:

```
gbdt_scores
```

输出:

```
[0.82122905027932958,
0.8314606741573034,
0.8202247191011236,
0.8258426966292135,
0.7696629213483146]
```

融合模型各个 folder 成绩:

```
stk_scores
```

输出:

```
[0.81564245810055869,
0.8146067415730337,
0.7865168539325843,
0.8146067415730337,
0.7752808988764045]
```

在不同的数据区间中，融合后的模型明显波动更小，说明融合后的模型对新数据集有更好的适应性。因为 titanic 样本数量很少，而每次数据集划分都有信息损失，所以我们在全部数据集中完成训练模型及融合后，在 Kaggle 的线上测试集中，确实可以看到 stacking 融合后的模型在未知数据集的成绩有一个小幅的提升：约 1% 左右。

从经验上看，在中小数据量级别的任务中，融合消除多个模型的缺陷，会使模型整体更健壮，这一方式对成绩的提升还是比较明显的。一般会选择几个实现思路差异程度较大的模型，如线性模型、SVM（如果训练时间允许）、随机森林、boosting 模型等。由于 titanic 任务中数据集过小，所以我们将模型融合与单个模型训练放在了同一数据集中如果数据量非常充分，那么更合理的做法是分为三个数据集，如图 2-25 所示。



训练单个模型

训练融合模型

测试

4 : 1 : 2

图 2-25 数据集分割比例

图 2-25 展示了一个仅供参考的分割方法，下面补充一下代码实现：

```
import numpy as np

def three_kfolder(data, n_folds=5, shuffle=True, ratios=[4, 1, 2]):
    """按 ratios 数组随机 (shuffle) 三分割数据集，
    返回: traing_set, stacking_set, testing_set"""
    assert ratios and len(ratios) == 3, 'ratios 必须是 3-items-arraylike 数组'
    data = np.array(data)
    N = len(data)
    ratios_nor = np.array(ratios).astype(float) / np.sum(ratios)
    ratios_num = (ratios_nor * N).astype(int).cumsum()

    for i in range(n_folds):
        ind = range(len(data))
        np.random.shuffle(ind)
        data_shuf = data[ind]
        yield data_shuf[:ratios_num[0]],
        data_shuf[ratios_num[0]:ratios_num[1]],
        data_shuf[ratios_num[1]:ratios_num[2]]

# 使用 demo
data = ['a', 'c', 'd', 'e', 'g', '2', 'f', 'c', '3', 'p']
for traing_set, stacking_set, testing_set in three_kfolder(data):
    print traing_set, stacking_set, testing_set
```

输出：

```
(array(['g', 'd', '2', 'a', 'c'],
      dtype='|S1'), array(['3'],
      dtype='|S1'), array(['f', 'c'],
      dtype='|S1'))
(array(['3', 'c', 'd', 'f', 'g'],
      dtype='|S1'), array(['p'],
      dtype='|S1'), array(['2', 'a'],
      dtype='|S1'))
(array(['c', 'c', 'a', 'g', 'd'],
      dtype='|S1'), array(['p'],
      dtype='|S1'), array(['2', '3'],
      dtype='|S1'))
(array(['a', 'g', 'p', '3', 'c'],
      dtype='|S1'), array(['c'],
      dtype='|S1'), array(['f', 'e'],
      dtype='|S1'))
```

```
(array(['a', 'e', 'f', '2', 'd'],
      dtype='<S1'), array(['g'],
      dtype='<S1'), array(['p', 'c'],
      dtype='<S1'))
```

## 2.6.5 泰坦尼克号生存预测：小结

回顾整个“泰坦尼克号生存预测”任务，工程实现基本分为五步：

- (1) 数据预处理。
- (2) 选择模型。
- (3) GridSearch 刷选模型预设参数。
- (4) 训练模型。
- (5) 调试，数据预处理优化、特征优化或者模型优化（更换模型 or 模型融合）。

在完成一个机器学习任务时，一般我们先用最简单、最快的方式实现一次（drop 无效数据后的逻辑分类），成绩不到 70%；然后又花了很多时间在数据和特征的优化上——这部分工作算“特征工程”，成绩有了可观的提升；提高又不断地使用复杂的模型努力提高成绩——这部分算“模型工程”，成绩提升了一点点；最后通过模型融合将之前的努力成果合在一起——这是我们目前能做到的最好成绩了。

下面给出一个描述基于经验下的各环节对任务最终成绩的提升比例饼图，如图 2-26 所示，具体任务下环节影响比例会不一样，此图仅供参考。

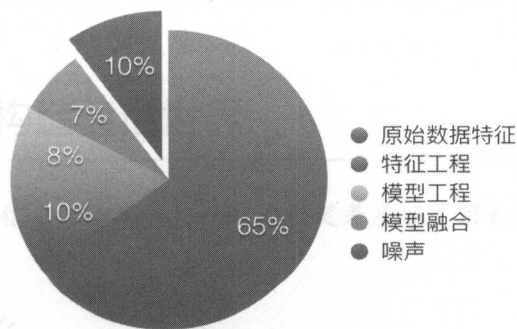


图 2-26 各环节提升总成绩比例

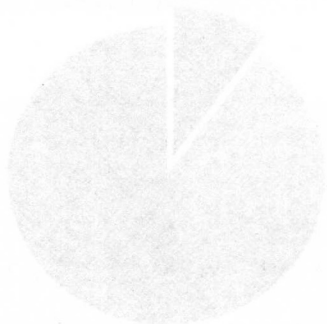
如图 2-26 所示，数据集中噪声的比例决定了机器学习模型的成绩上限。

对于中小级别数据量的机器学习任务，解决问题的经历其实和“泰坦尼克号生存预测”任务一样，对模型最终成绩的影响因素的排序是：原始的数据质量、特征质量 > 特

征工程 > 模型工程、模型融合。而对于海量数据级别的机器学习任务，数据质量、特征质量决定了绝大部分，优质特征下的海量级别数据，一个简单的模型就可以比较接近模型能做到的最好成绩了。

## 2.6.6 回顾

- 群体的智慧：消除每个模型个体的“个性”，去过拟合，让整体表现更好。
- Bagging：构造数据集的随机子集训练模型个体，等权重投票融合模型个体的预测。
- Boosting：生成模型个体序列  $M(M_1, M_2, \dots, M_n)$ ，后位的模型尝试修正前面模型的错误。
- Stacking：通过融合不同类型弥补模型本身的先天缺陷。
  - 加权投票，见 `sklearn.ensemble.VotingClassifier`。
  - 用新的线性模型融合。
  - 合理使用数据集，注意防止信息泄露。



# 实战：股票量化

当使用者把机器学习技术作为工具落地到其他领域时，往往会陷入极左或者极右的境地。

比如在把机器学习运用到股票市场时，很多朋友私下问我：“应该拿什么特征预测股票价格？”极左的人认为机器学习技术已经能预测整个市场，他们往往在得到一个肯定的回答之后就会把这个工具技术视作一切。而极右的人则会问：“机器学习技术能对股票市值/技术分析有什么帮助？”。在他们的金融生态系统里，看不到机器学习的用武之地。

在我看来，机器学习代表的远不仅仅是数据自动化的高效工具技术，更是一种科学的数据思考方式。在将机器学习技术落地到其他领域时，我们需要做的是，要把数据科学的思考方式和实现技术有机地融入新领域，产生新的模式。

本章，我们将以温习前面两章的知识点为目的，初步探索“将机器学习落地到股票量化”这一课题。本章会涉及一些金融股票概念，由于篇幅有限不好展开，大家可适当略过，关注核心思想即可。

## 3.1 第一步：构造童话世界

孩子理直气壮地相信童话和梦想，而我们微笑着质疑，用我们自以为是沧桑。

——寂地 MY WAY

### 3.1.1 股票是什么

一只股票就是一个股份制公司的所有权，股份制公司通过对其所有权的交易，进行快速融资，获得资本后快速扩张。获得公司所有权的股民可以和公司共享利润，简单地说，当你买入一只股票后你就是这家公司的一个股东！

在股票市场中，股民可以在交易市场买卖股票，获得差价利润/损失。交易所实时统

计交易的数据，在交易软件终端反馈给股民。对于一只股票，我们关心其中的一些统计的“量价”数据（以“天”为单位）。

- 开盘价：股市当天开始交易时，一只股票的价格。
- 收盘价：股市当天停止交易时，一只股票的价格。
- 成交量：一只股票当天的买卖换手数量。

### 3.1.2 当机器学习与量化交易走在一起

我们的任务场景是证券机构提供股票市场“过去”的数据，而股民的目标是“未来”的交易机会。当机器学习落地在股票市场时，很多人容易过度膨胀机器学习的能力，将现实错综复杂的因素看得格外简单，他们眼里的实现方式是这样的：

机器学习.fit(x, y) == (股价预测，涨跌预测) == 发财

Ok，先满足这种幻觉，再进入童话世界寻找这种可能。接下来，我们将分为三步验证这种幻觉是否成立。

(1) 构造一个“童话版”世界中的股票的走势，股票背后有一个简单客观的规律控制其走势，这个规律公式是我们机器学习希望通过拟合数据找到的“黑箱”知识。

(2) 在童话世界中，我们运用机器学习发现股票背后的规律，并对比机器学习发现的规律和股票背后真实的规律。

(3) 把童话世界中验证可行的实现方式放到真实世界会发生什么？

### 3.1.3 构造一个童话世界

下面的代码通过 `change_real_to_another_word()` 将现实世界中的股票数据转换收盘价格后变成童话世界中的股票收盘价格，核心代码在 `_gen_another_word_price_rule()` 中。通过前天收盘量价、昨天收盘量价和今天的量，构建了另一个世界中的价格模型。

构建童话世界的股票走势：

```
import numpy as np
from abupy import ABUSymbolPd

def _gen_another_word_price(kl_another_word):
    """
    生成股票在另一个世界中的价格
    :param kl_another_word:
    :return:
    """
```



```

for ind in np.arange(2, kl_another_word.shape[0]):
    # 前天数据
    bf_yesterday = kl_another_word.iloc[ind - 2]
    # 昨天数据
    yesterday = kl_another_word.iloc[ind - 1]
    # 今天数据
    today = kl_another_word.iloc[ind]
    # 生成今天的收盘价格
    kl_another_word.close[ind] = _gen_another_word_price_rule(
        yesterday.close, yesterday.volume,
        bf_yesterday.close, bf_yesterday.volume,
        today.volume, today.date_week)

def _gen_another_word_price_rule(yesterday_close, yesterday_volume,
                                  bf_yesterday_close,
                                  bf_yesterday_volume,
                                  today_volume, date_week):
    """
    通过前天收盘量价、昨天收盘量价和今天的量，构建另一个世界中的价格模型
    """
    # 昨天收盘价格与前天收盘价格的价格差
    price_change = yesterday_close - bf_yesterday_close
    # 昨天成交量与前天成交量的量差
    volume_change = yesterday_volume - bf_yesterday_volume
    # 如果量和价变动一致，则今天价格上涨，否则下跌
    # 即量价齐涨→涨，量价齐跌→跌，量价不一致→跌
    sign = 1.0 if price_change * volume_change > 0 else -1.0

    # 针对 sign 生成噪音，噪音生效的先决条件是今天的量是这三天最大的
    gen_noise = today_volume > np.max(
        [yesterday_volume, bf_yesterday_volume])
    # 如果量是这三天最大，且是周五，下跌
    if gen_noise and date_week == 4:
        sign = -1.0
    # 如果量是这三天最大，且是周一，上涨
    elif gen_noise and date_week == 0:
        sign = 1.0
    # 今天的涨跌幅度基础是 price_change（昨天和前天的价格变动）
    price_base = abs(price_change)
    # 今天的涨跌幅度变动因素：量比，
    # “今天的成交量/昨天的成交量” 和 “今天的成交量/前天的成交量” 的均值
    price_factor = np.mean([today_volume / yesterday_volume,
                             today_volume / bf_yesterday_volume])
    if abs(price_base * price_factor) < yesterday_close * 0.10:
        # 如果 量比 * price_base 没超过 10%，今天价格计算
        today_price = yesterday_close + \
            sign * price_base * price_factor
    else:

```



```

# 如果涨跌幅度超过 10%，限制上限，下限为 10%
today_price = yesterday_close + sign * yesterday_close * 0.10
return today_price

def change_real_to_another_word(symbol):
    """
    将原始真正的股票数据价格列只保留前两天数据，成交量、周几列完全保留，
    价格列其他数据使用_gen_another_word_price，变成另一个世界的价格
    :param symbol:
    :return:
    """
    kl_pd = ABUSymbolPd.make_kl_df(symbol)
    if kl_pd is not None:
        # 原始股票数据也只保留价格、周几、成交量
        kl_fairy_tale = kl_pd.filter(['close', 'date_week', 'volume'])
        # 只保留头两天的原始交易收盘价格，其他的都赋予 nan
        kl_fairy_tale['close'][2:] = np.nan
        # 将其他 nan 价格变成童话世界中的价格需使用_gen_another_word_price
        _gen_another_word_price(kl_fairy_tale)
        return kl_fairy_tale

```

下面的代码选定了一些股票，并使用 `change_real_to_another_word()` 函数构建童话世界的价格走势，从输出可以看到生成的数据新走势，数据包括：收盘价、周几、成交量。

```

# 选择若干美股股票
choice_symbols = ['usNOAH', 'usSFUN', 'usBIDU', 'usAAPL', 'usGOOG',
                  'usTSLA', 'usWUBA', 'usVIPS']
another_word_dict = {}
real_dict = {}
for symbol in choice_symbols:
    # 童话世界的股票走势字典
    another_word_dict[symbol] = change_real_to_another_word(symbol)
    # 真实世界的股票走势字典，这里不考虑运行效率问题
    real_dict[symbol] = ABUSymbolPd.make_kl_df(symbol)
# 表 3-1 所示
another_word_dict['usNOAH'].head()

```

输出如表 3-1 所示。

表 3-1 童话世界中 usNOAH 股票数据采样

	close	date_week	volume
2014-07-24	15.210000	3	307211
2014-07-25	15.320000	4	101442
2014-07-28	15.753858	0	601568
2014-07-29	17.329244	1	655297
2014-07-30	18.204088	2	348344

对比真实的 noah 数据：

# 表 3-2 所示

```
real_dict['usNOAH'].head().filter(['close', 'date_week', 'volume'])
```

输出如表 3-2 所示。

表 3-2 真实世界中 usNOAH 股票数据采样

	close	date_week	volume
2014-07-24	15.21	3	307211
2014-07-25	15.32	4	101442
2014-07-28	16.13	0	601568
2014-07-29	16.75	1	655297
2014-07-30	16.83	2	348344

可以发现收盘价格的 2014-7-24 和 2014-7-25 是相同的，但之后的价格就不相同了，date\_week 和 volume 数据都是一致的。

接下来对比真实世界与生成的童话世界中的股票走势图，由于不涉及随机概率元素，所以生成的另一个世界的股价走势是固定的，即读者在 IPython Notebook 环境下运行本书代码，生成的运行走势也将与本书一致，结果如图 3-1 所示。

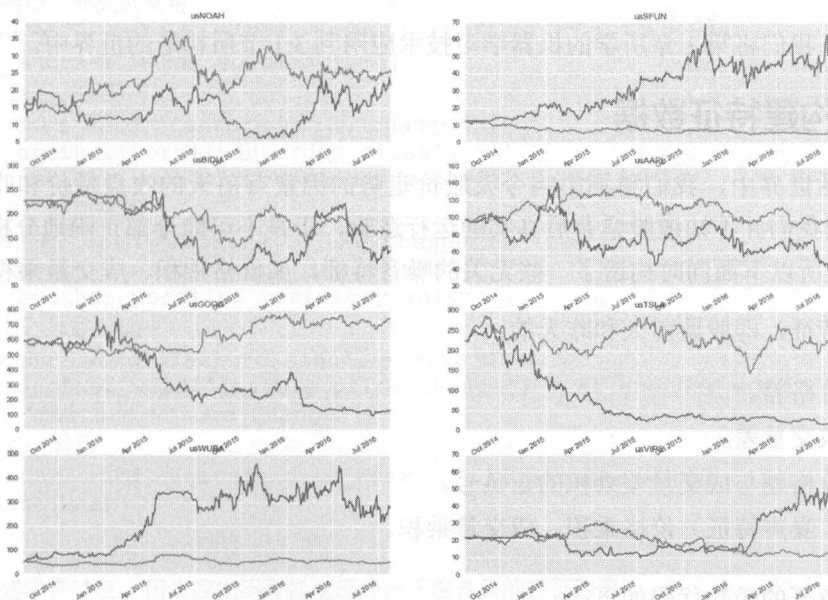


图 3-1 股票走势图对比

备注：彩图见随书 Git 库示例代码运行结果。

```
import itertools
# 4 * 2
_, axs = plt.subplots(nrows=4, ncols=2, figsize=(20, 15))
# 将画布序列拉平
axs_list = list(itertools.chain.from_iterable(axs))
for symbol, ax in zip(choice_symbols, axs_list):
    # 绘制童话世界的股价走势
    another_word_dict[symbol].close.plot(ax=ax)
    # 同样的股票在真实世界的股价走势
    real_dict[symbol].close.plot(ax=ax)
    ax.set_title(symbol)
```

### 3.1.4 回顾

本节构造了一个客观规律更简单的童话世界，方便后面小节使用机器学习技术和真实规律做结果对比。

## 3.2 第二步：应用机器学习

理论脱离实践是最大的不幸。

——达·芬奇

本节，我们将第1章所学的机器学习技术应用到3.1节所构造的世界中。

### 3.2.1 构建特征数据

在童话世界中，我们猜测影响今天股价走势的因素有前天的收盘量价和昨天的收盘量价。因为我们并不知道股票背后真实的运行逻辑，于是不可能全部正确地分析出真实的特征因素，所以下面同时构造了一些无关的噪音特征，如价格乘积、成交量乘积。

输入特征，即股票前天和昨天的：

- 价格差。
- 成交量差。
- 价格差与成交量差乘积的正负号。
- （噪声特征）价格乘积、成交量乘积。

构建特征的函数代码如下：

```

def gen_fairy_tale_feature(kl_another_word):
    """
    构建特征模型函数
    生成的 dataframe 有收盘价、周几、成交量列
    """
    # y 值使用 close.pct_change, 即涨跌幅度
    kl_another_word['regress_y'] = kl_another_word.close.pct_change()
    # 前天收盘价格
    kl_another_word['bf_yesterday_close'] = 0
    # 昨天收盘价格
    kl_another_word['yesterday_close'] = 0
    # 昨天收盘成交量
    kl_another_word['yesterday_volume'] = 0
    # 前天收盘成交量
    kl_another_word['bf_yesterday_volume'] = 0
    # 对齐特征, 即前天收盘价与今天的收盘价错开 2 个时间单位, [2:] =[:-2]
    kl_another_word['bf_yesterday_close'][2:] = \
        kl_another_word['close'][:-2]
    # 对齐特征, 前天成交量
    kl_another_word['bf_yesterday_volume'][2:] = \
        kl_another_word['volume'][:-2]
    # 对齐特征, 昨天收盘价与今天收盘价错开 1 个时间单位, [1:] =[:-1]
    kl_another_word['yesterday_close'][1:] = \
        kl_another_word['close'][:-1]
    # 对齐特征, 昨天成交量
    kl_another_word['yesterday_volume'][1:] = \
        kl_another_word['volume'][:-1]
    # 特征 1: 价格差
    kl_another_word['feature_price_change'] = \
        kl_another_word['yesterday_close'] - \
        kl_another_word['bf_yesterday_close']
    # 特征 2: 成交量差
    kl_another_word['feature_volume_Change'] = \
        kl_another_word['yesterday_volume'] - \
        kl_another_word['bf_yesterday_volume']
    # 特征 3: 涨跌 sign
    kl_another_word['feature_sign'] = np.sign(
        kl_another_word['feature_price_change'] * kl_another_word[
            'feature_volume_Change'])
    # 特征 4: 周几
    kl_another_word['feature_date_week'] = kl_another_word[
        'date_week']

    """
    构建噪音特征, 因为我们不可能全部分析正确真实的特征因素
    所以这里引入一些噪音特征
    """

```



```

# 成交量乘积
kl_another_word['feature_volume_noise'] = \
    kl_another_word['yesterday_volume'] * \
    kl_another_word['bf_yesterday_volume']
# 价格乘积
kl_another_word['feature_price_noise'] = \
    kl_another_word['yesterday_close'] * \
    kl_another_word['bf_yesterday_close']
# 将数据标准化
scaler = preprocessing.StandardScaler()
kl_another_word['feature_price_change'] = scaler.fit_transform(
    kl_another_word['feature_price_change'])
kl_another_word['feature_volume_change'] = scaler.fit_transform(
    kl_another_word['feature_volume_change'])
kl_another_word['feature_volume_noise'] = scaler.fit_transform(
    kl_another_word['feature_volume_noise'])
kl_another_word['feature_price_noise'] = scaler.fit_transform(
    kl_another_word['feature_price_noise'])
# 只筛选 feature_ 开头的特征和 regress_y, 抛弃前两天数据, 即[2:]
kl_fairy_tale_feature = kl_another_word.filter(
    regex='regress_y|feature_*')[2:]
return kl_fairy_tale_feature

```

使用之前生成的童话世界股票的走势数据构建训练集特征模型:

```

fairy_tale_feature = None
for symbol in another_word_dict:
    # 首先拿出对应的走势数据
    kl_another_word = another_word_dict[symbol]
    # 通过走势数据生成训练集特征
    kl_feature = gen_fairy_tale_feature(kl_another_word)
    # 将每个股票的特征数据都拼接起来, 形成训练集
    fairy_tale_feature = kl_feature if fairy_tale_feature is None \
        else fairy_tale_feature.append(kl_feature)

```

查看训练集数据:

```

print fairy_tale_feature.shape
fairy_tale_feature.tail()

```

输出:

```
(4016, 7)
```

童话世界特征数据采样如表 3-3 所示。

表 3-3 童话世界特征数据采样

	regress_y	feature_price _change	feature_volume _Change	feature_sig n	feature_date _week	feature_volume _noise	feature_price_n oise
2016-07-20	0.050111	2.848368	1.584608	1	2	0.284525	2.024521
2016-07-21	-0.026259	1.540837	-1.830352	-1	3	0.083452	2.539770
2016-07-22	0.013823	-0.948895	-0.146223	1	4	-0.440273	2.626239
2016-07-25	-0.011446	0.387842	-0.319326	-1	0	-0.542684	2.576028
2016-07-26	-0.000669	-0.445437	0.030107	-1	1	-0.578196	2.584625

### 3.2.2 回归预测股票价格

目标：使用昨天和前天的特征，回归预测股票今天的股价。

首先使用上述数据作为训练集，要想使用回归训练数据，则需要  $X$  特征矩阵和连续值  $y$  序列，构建过程如下：

```
# Dataframe → matrix
feature_np = fairy_tale_feature.as_matrix()
# X特征矩阵
train_x = feature_np[:, 1:]
# 回归的连续值 y
train_y_regress = feature_np[:, 0]
# 分类训练的离散值 y，之后分类技术使用
train_y_classification = np.where(train_y_regress > 0, 1, 0)

train_x[:5], train_y_regress[:5], train_y_classification[:5]
```

输出：

```
(array([[0.07439263, -0.11774155, -1., 0., -0.28811183, -0.62390726],
        [0.09901398, 0.59219085, 1., 1., -0.10955316, -0.59749127],
        [0.0845932, -0.61666006, -1., 2., -0.11727893, -0.56967334],
        [-0.10418886, -0.5139319, 1., 3., -0.52956586, -0.56547335],
        [0.08850743, 1.1952326, 1., 4., -0.33584097, -0.56083711]]),
array([0.00813274, 0.00714915, -0.00483353, 0.00738148, 0.00851086]),
array([1, 1, 0, 1, 1]))
```

下面使用一个不在训练集中的股票 usFB 生成需要的测试集，通过 `gen_feature_from_symbol()` 函数封装上述零散操作，生成测试集数据：

```
def gen_feature_from_symbol(symbol):
    """
    封装由一个 symbol 转换为特征矩阵的序列函数
    """
    # 真实世界走势数据转换到童话世界
```



```

kl_another_word = change_real_to_another_word(symbol)
# 由走势转换为特征 dataframe
kl_another_word_feature_test = \
    gen_fairy_tale_feature(kl_another_word)
# 转换为 matrix
feature_np_test = kl_another_word_feature_test.as_matrix()
# 从 matrix 抽取 y 回归
test_y_regress = feature_np_test[:, 0]
# y 回归 → y 分类
test_y_classification = np.where(test_y_regress > 0, 1, 0)
# 从 matrix 抽取 x 特征矩阵
test_x = feature_np_test[:, 1:]
return test_x, test_y_regress, test_y_classification, \
    kl_another_word_feature_test

test_x, test_y_regress, test_y_classification, \
    kl_another_word_feature_test = gen_feature_from_symbol('usFB')

```

下面使用 `scikit-learn` 的线性回归模块预测股价涨跌幅度，图 3-2 所示的线性回归基本可以完美预测，通过 `cross_validation.cross_val_score()` 函数进行交叉训练验证，计算 RMSE 数值作为预测准确度的度量标准。

```

from sklearn.linear_model import LinearRegression
from sklearn import cross_validation

def regress_process(estimator, train_x, train_y_regress, test_x,
                    test_y_regress):
    # 训练训练集数据
    estimator.fit(train_x, train_y_regress)
    # 使用训练好的模型预测测试集对应的 y，即根据 usFB 的走势特征预测股价
    test_y_predict_regress = estimator.predict(test_x)

    # 绘制 usFB 实际股价
    plt.plot(test_y_regress.cumsum())
    # 绘制通过模型预测的 usFB 股价
    plt.plot(test_y_predict_regress.cumsum())

    # 针对训练集数据做交叉验证
    scores = cross_validation.cross_val_score(estimator, train_x,
                                                train_y_regress, cv=10,
                                                scoring=
                                                    'mean_squared_error')

    # mse 开方 → rmse
    mean_sc = np.mean(np.sqrt(-scores))
    print('RMSE: ' + str(mean_sc))

# 实例化线性回归对象 estimator
estimator = LinearRegression()

```

```
# 将回归模型对象、训练集 x、训练集连续 y 值、测试集 x、测试集连续 y 值传入
regress_process(estimator, train_x, train_y_regress, test_x,
                 test_y_regress)
```

输出如图 3-2 所示。

```
RMSE: 0.0260964344834
```

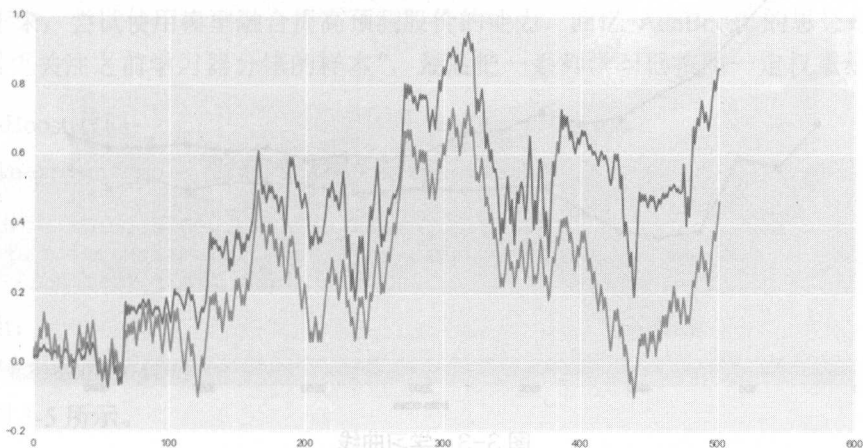


图 3-2 线性回归模块预测股价涨跌幅度

核心思想：本章开始编写的另一个世界股价走势代码 `_gen_another_word_price_rule()` 函数对童话世界的人来说是个黑箱，实际上我们并不知道那个函数里面到底是怎么运行的。它可以说是一个上帝函数，虽然我们无法通过编写 `rule` 的方式来实现预测股价，但经过反复推敲测试，寻找历史上与股价最相关的一些特征，编写了提取这些特征的代码 `gen_fairy_tale_feature()` 函数，然后运用机器学习的算法反向倒推出模型算法，也就是达到如下效果：

```
fairy_tale_estimator = 机器学习.fit(gen_fairy_tale_feature)
```

```
fairy_tale_estimator.predict(one_stock) ≈ _gen_another_word_price_rule(one_stock)
```

主要思想已讲完，下面将使用机器学习的一些其他算法来示例。算法的好坏并不那么重要，特别是在数据量足够多的情况下，特征的选取与数据的获取才是工程上最主要的问题。

下面通过 `abu` 量化系统中的 `ABuMLExecute` 来绘制学习曲线，观察随着样本数增大，算法在训练集和测试集的表现如何。如下所示，具体源代码请查阅 `ABuMLExecute.py`。

学习曲线：

```
from abupy import ABuMLExecute
```

```
ABuMLExecute.plot_learning_curve(estimator, train_x, train_y_regress,
                                  cv=10)
```

如图 3-3 所示。

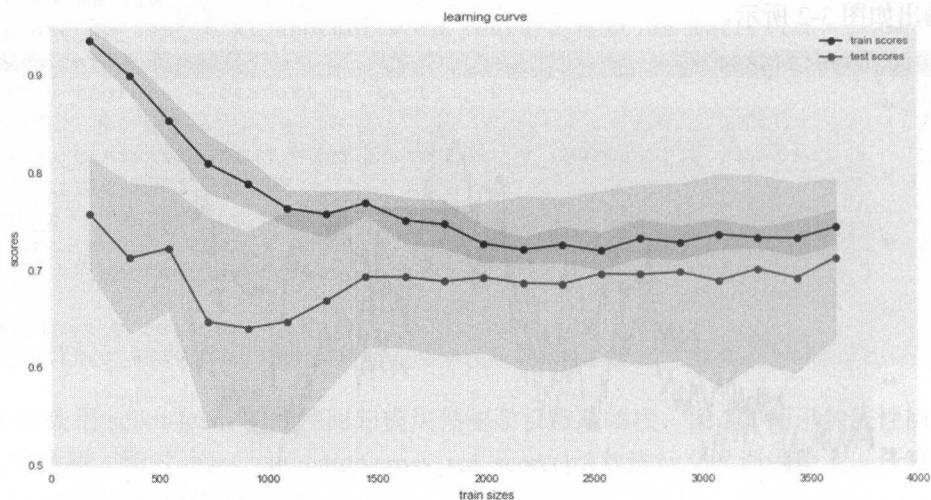


图 3-3 学习曲线

下面的代码使用多项式来提高拟合程度，可以看到 RMSE 变得更小了，如图 3-4 所示。

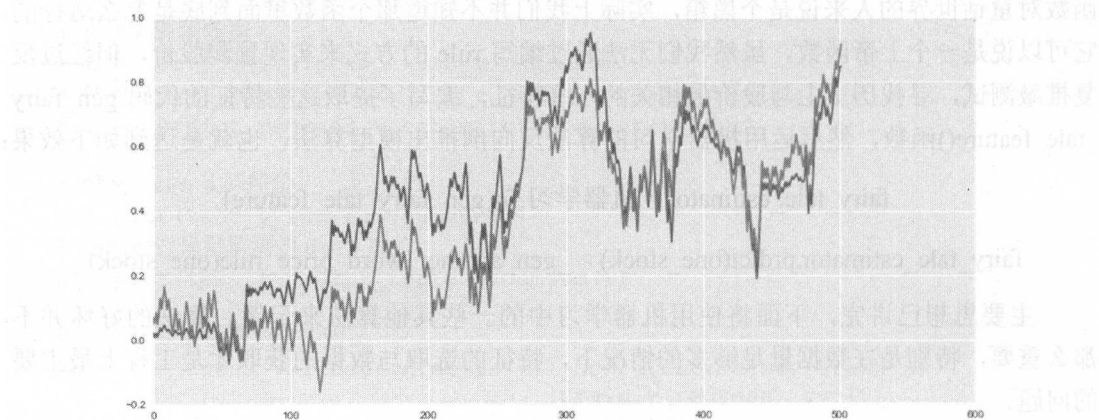


图 3-4 degree=3 预测股价涨跌幅度

多项式回归：

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import PolynomialFeatures
# pipeline 套上 degree=3 + LinearRegression
estimator = make_pipeline(PolynomialFeatures(degree=3),
```

```
LinearRegression())
# 继续使用 regress_process, 区别是 estimator 变了
regress_process(estimator, train_x, train_y_regress, test_x,
                 test_y_regress)
```

输出:

```
RMSE: 0.0242783959238
```

接下来, 尝试使用模型融合提高预测股价的能力, 回忆 AdaBoost 的思想是“使下一个学习器更关注之前学习器分错的样本”, 最后把一系列学习器按照一定权重组合在一起。

AdaBoost 代码:

```
from sklearn.ensemble import AdaBoostRegressor

estimator = AdaBoostRegressor(n_estimators=100)
regress_process(estimator, train_x, train_y_regress, test_x,
                 test_y_regress)
```

输出:

```
RMSE: 0.0236202304171
```

如图 3-5 所示。

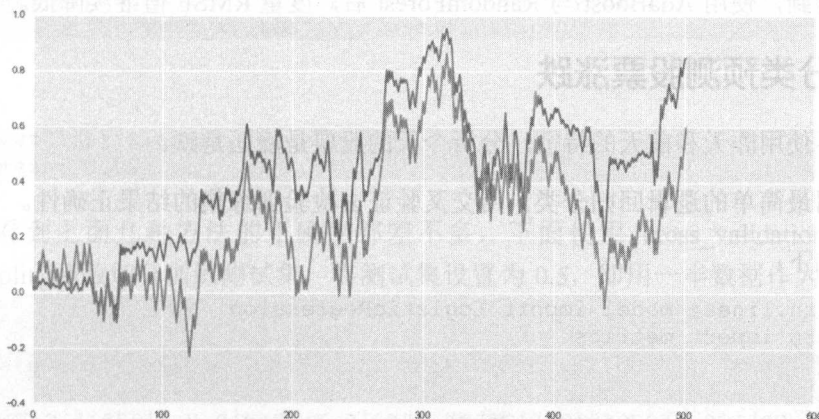


图 3-5 AdaBoost 预测股价涨跌幅度

而随机森林就是“一系列决策树模型组合起来的投票模型”。

随机森林代码:

```
from sklearn.ensemble import RandomForestRegressor

estimator = RandomForestRegressor(n_estimators=100)
regress_process(estimator, train_x, train_y_regress, test_x,
                 test_y_regress)
```







```
# 针对训练集数据做交叉验证 scoring='accuracy'
scores = cross_validation.cross_val_score(estimator, train_x,
                                          train_y_classification,
                                          cv=10,
                                          scoring='accuracy')

mean_sc = np.mean(scores)
print('cross validation accuracy mean: {:.2f}'.format(mean_sc))

estimator = LogisticRegression(C=1.0, penalty='l1', tol=1e-6)
# 将分类器、训练集 x、训练集 y 分类，测试集、测试集 y 分类分别传入函数
classification_process(estimator, train_x, train_y_classification,
                      test_x, test_y_classification)
```

输出:

```
LogisticRegression accuracy = 0.93
accuracy mean: 0.92
```

使用随机森林来运行并查看效果:

```
from sklearn.ensemble import RandomForestClassifier

estimator = RandomForestClassifier(n_estimators=100)
classification_process(estimator, train_x, train_y_classification,
                      test_x, test_y_classification)
```

输出:

```
RandomForestClassifier accuracy = 0.93
accuracy mean: 0.92
```

可以看到上面几种方法的正确率都差不多，下面使用 `cross_validation` 模块下的 `train_test_split()` 函数切分训练测试集，将测试集设置为 0.5，即用一半数据作为测试集，来看看效果。

```
from sklearn.cross_validation import train_test_split

def train_test_split_xy(estimator, x, y, test_size=0.5,
                       random_state=0):
    # 通过 train_test_split 将原始训练集随机切割为新训练集与测试集
    train_x, test_x, train_y, test_y = \
        train_test_split(x, y, test_size=test_size,
                        random_state=random_state)

    print(x.shape, y.shape)
    print(train_x.shape, train_y.shape)
    print(test_x.shape, test_y.shape)

    clf = estimator.fit(train_x, train_y)
    predictions = clf.predict(test_x)
```

```

# 度量准确率
print("accuracy = %.2f" %
      (metrics.accuracy_score(test_y, predictions)))

# 度量查准率
print("precision_score = %.2f" %
      (metrics.precision_score(test_y, predictions)))

# 度量回收率
print("recall_score = %.2f" %
      (metrics.recall_score(test_y, predictions)))

return test_y, predictions

test_y, predictions = train_test_split_xy(estimator, train_x,
                                          train_y_classification)

```

输出:

```

((4016, 6), (4016,))
((2008, 6), (2008,))
((2008, 6), (2008,))
accuracy = 0.92
precision_score = 0.93
recall_score = 0.92

```

可以看到，即便使用一半数据作为训练集，另一半数据作为测试集，正确率分数、查准率分数及召回率分数也都还在 0.92 以上。下面使用 `metrics.confusion_matrix()` 函数来显示混淆矩阵情况，获得数据样本表现的二维分布。用 `metrics.classification_report()` 函数用来查看其他分类信息。

```

def confusion_matrix_with_report(test_y, predictions):
    confusion_matrix = metrics.confusion_matrix(test_y, predictions)
    # print("Confusion Matrix ", confusion_matrix)
    print("          Predicted")
    print("          | 0 | 1 |")
    print("          |----|----|")
    print("          0 | %3d | %3d |" % (confusion_matrix[0, 0],
                                confusion_matrix[0, 1]))
    print("Actual    |----|----|")
    print("          1 | %3d | %3d |" % (confusion_matrix[1, 0],
                                confusion_matrix[1, 1]))
    print("          |----|----|")

    print(metrics.classification_report(test_y, predictions))

confusion_matrix_with_report(test_y, predictions)

```

输出：

Predicted							
	0	1					
Actual	0	903	73				
	1	84	948				
		precision	recall	f1-score	support		
	0	0.91	0.93	0.92	976		
	1	0.93	0.92	0.92	1032		
avg / total		0.92	0.92	0.92	2008		

可以看到二三象限的假阳、假阴大概占比 10%，分布均衡。

下面使用 ROC 曲线看一下效果，如图 3-7 所示，具体绘制代码请查阅 ABuMLExecute。

绘制 ROC 曲线：

```
ABuMLExecute.plot_roc_estimator(estimator, train_x,
                                train_y_classification)
```

输出如图 3-7 所示。

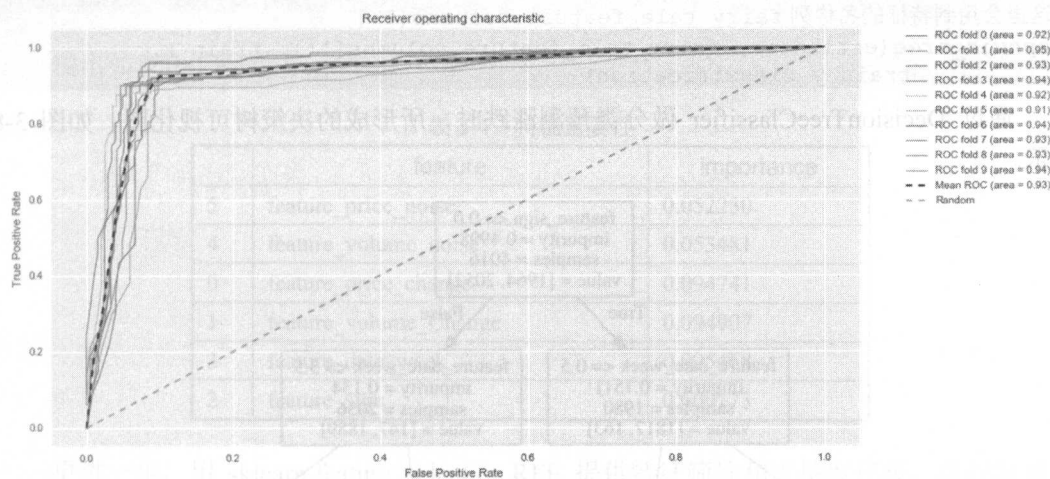


图 3-7 ROC 曲线

### 3.2.4 通过决策树分类，绘制决策图

回忆决策树是通过不断寻找最佳分割特征建树，从而完成分类/回归目的的一类模型。在工程应用中，这个模型的另一个好处是可以绘制直观的决策图，帮助我们研究问题本身的一些特性。通过 `DecisionTreeClassifier` 等支持 `tree` 系的分类算法，使用 `pydot` 和 `graphviz` 包可以绘制决策图，代码如下所示。

```
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree
import os

estimator = DecisionTreeClassifier(max_depth=2, random_state=1)

def graphviz_tree(estimator, features, x, y):
    if not hasattr(estimator, 'tree_'):
        print('only tree can graphviz!')
        return

    estimator.fit(x, y)
    # 将决策模型导出成 graphviz.dot 文件
    tree.export_graphviz(estimator.tree_, out_file='graphviz.dot',
                        feature_names=features)
    # 通过 dot 将模型绘制成决策图，保存 png
    os.system("dot -T png graphviz.dot -o graphviz.png")

# 这里会用到特征的名称列 fairy_tale_feature.columns[1:]
graphviz_tree(estimator, fairy_tale_feature.columns[1:], train_x,
              train_y_classification)
```

通过 `DecisionTreeClassifier` 做分类预测涨跌时，所形成的决策树可视化图，如图 3-8 所示。

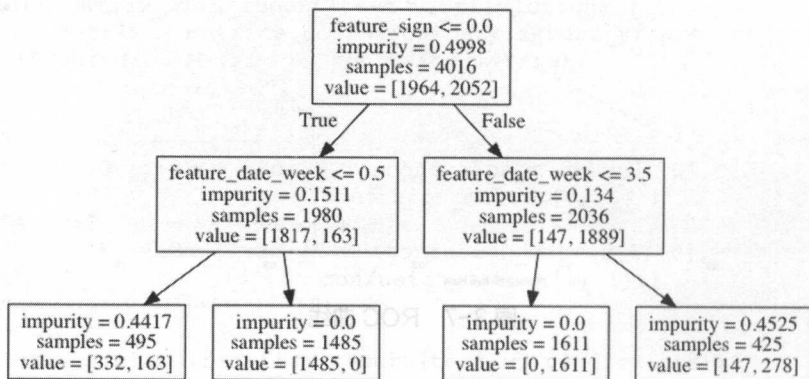


图 3-8 决策树分类



`gen_fairy_tale_feature()`函数中仍然存在两个噪音特征：成交量乘积与价格乘积。我们如何选取特征呢？下面代码将使用 `feature_importances_` 或者 `coef_` 来查看特征的重要程度。

下面使用 `RandomForestClassifier` 作为示例分类器来查看特征的重要性。从输出可以发现，新添加的两个噪音权重的重要程度为所有特征中最小的两个。

```
def importances_coef_pd(estimator):
    if hasattr(estimator, 'feature_importances_'):
        # 有 feature_importances_ 的通过 sort_values 排序
        return pd.DataFrame(
            {'feature': list(fairy_tale_feature.columns[1:]),
             'importance': estimator.feature_importances_}).sort_values(
                'importance')
    elif hasattr(estimator, 'coef_'):
        # 有 coef_ 的通过 coef 排序
        return pd.DataFrame(
            {"columns": list(fairy_tale_feature.columns)[1:],
             "coef": list(estimator.coef_.T)}).sort_values(
                'coef')
    else:
        print('estimator not hasattr feature_importances_ or coef_!')
# 使用随机森林分类器
estimator = RandomForestClassifier(n_estimators=100)
# 训练数据模型
estimator.fit(train_x, train_y_classification)
# 对训练后的模型特征的重要程度进行判定，重要程度由小到大，如表 3-4 所示
importances_coef_pd(estimator)
```

输出如表 3-4 所示。

表 3-4 特征重要性

	feature	importance
5	feature_price_noise	0.052230
4	feature_volume_noise	0.053481
0	feature_price_change	0.094741
1	feature_volume_Change	0.094907
3	feature_date_week	0.095468
2	feature_sign	0.609173

更进一步，用 `sklearn.feature_selection.RFE` 提供特征筛选和支持度评级，可以发现，新添加的两个噪音的支持度排在最后两位，代码如下所示。

```
from sklearn.feature_selection import RFE
```



```
def feature_selection(estimator, x, y):
    selector = RFE(estimator)
    selector.fit(x, y)
    print('RFE selection')
    print(pd.DataFrame(
        {'support': selector.support_, 'ranking': selector.ranking_,
         index=fairy_tale_feature.columns[1:]})
    )
feature_selection(estimator, train_x, train_y_classification)
```

输出:

```
RFE selection
           ranking support
feature_price_change      1   True
feature_volume_Change     1   True
feature_sign              1   True
feature_date_week         2  False
feature_volume_noise      3  False
feature_price_noise       4  False
```

这就是机器学习在童话世界中的历程。由此可以发现，在童话世界中使用机器学习技术，可以做到对股价的小误差预测以及对涨跌的概率预测。也就是说，在简单因素的股票市场中利用机器学习是可以战胜市场的。

### 3.2.5 回顾

在童话世界中，用机器学习直接拟合背后的运行规律，效果不错。

## 3.3 第三步：在真实世界应用机器学习

幻想都是用来破灭的，童话总会在现实面前变成普通话。而成熟的人们，却永不甘心绝望……

——《花儿与少年》

在真实世界的股票市场中，如果按照之前的实现方式构建模型，那么结局一定是忧伤的。为什么呢？

在童话中设定的可以影响股价走势的参数个数是有限的，特别是影响涨跌的因素，很容易构造特征。但在真实市场中，影响股价走势的因素有无限多个，而且这些因素之间也可以是相关的。就像你求解一个方程组，这个方程组不是有一两个解，而有无限个解，并且每个解都与其他任意解相关，但又并非简单的线性相关。《人类简史》中提到过，市场是一个二级混沌系统，所以可以认为任何想通过技术对股价进行预测或者涨跌预测都是

不可能的，不论你自己认为你使用的技术本身有多高深，也都无异于管中窥豹。本节，让我们从梦中醒来。

先来回顾一下童话世界的可喜战绩，代码如下，abupy 中封装了 scikit-learn 中很多常用的方法。

```
from abupy import AbuML
# 通过 X, Y 矩阵和特征的 DataFrame 对象组成 AbuML
ml = AbuML(train_x, train_y_classification, fairy_tale_feature)
# 使用随机森林作为分类器
_ = ml.estimator.random_forest_classifier()

# 交织验证结果的正确率
ml.cross_val_accuracy_score()
# 特征的选择
ml.feature_selection()
```

输出：

```
accuracy mean: 0.91883391499
RFE selection
```

	ranking	support
feature_price_change	1	True
feature_volume_Change	1	True
feature_sign	1	True
feature_date_week	2	False
feature_volume_noise	3	False
feature_price_noise	4	False

AbuML 包含了绘制学习曲线、roc 曲线、决策边界、混淆矩阵等常用方法，由于篇幅所限，这里不过多展开，详情请查询 Git 库 <https://github.com/bbfamily/abu> 或者公众号:abu\_quant 中的 abu 文档。

### 3.3.1 回测

回测是指当我们完成某种交易的模型（策略）后，将其运用到历史数据中进行模拟交易并观察收益结果的过程。下面将在真实世界的股票市场中，使用 3.2 节的机器学习方法试水。

abu 量化系统支持在回测过程中生成特征数据，切分训练测试集，甚至成交买单快照图片。

- `g_enable_ml_feature` 将在生成最终的输出结果数据 `orders_pd` 中加上买入时刻的很多信息，比如价格位置、趋势走向、波动情况等，核心实现是

AbuFactorBuyBase 中的 `make_buy_order_ml_feature()` 方法，该方法在每次生成 order 时被调用。

- `env.g_enable_train_test_split` 将选定的股票池切割成两部分：回测训练集与回测试集。其内部实现使用 `sklearn.cross_validation.KFold` 来打散切分数据，详情请查询 `ABUSymbol.market_train_test_split()` 函数。

如果只是在系统中使用上述功能则很简单，只需在 `abupy.env` 中做一些回测全局设置即可，设置代码如下。

```
abupy.env.g_enable_ml_feature = True
abupy.env.g_enable_train_test_split = True
```

因子在量化领域是指决策的因素。我们使用  $N$  日突破买入因子和三个卖出因子，不使用选股因子，`choice_symbols` 为 `None`。下面进行全市场策略回测，假设初始化资金为 200 万元，资金管理依然使用默认的 `atr`，每笔交易的买入基数资金设置为万分之十五 (`abupy.beta.atr.g_atr_pos_base = 0.0015`)。这个值如果设置得太大，比如初始默认的 0.1，则将会导致太多的股票由于资金不足而无法买入，丧失全市场回测的意义。如果设置得太小，就又会导致资金利用率下降（如下度量结果中资金利用率为 86.22%），使得最终收益下降。度量类中的策略买入成交比例就是对这个值进行度量的指标（如下度量结果中显示为 32.67%），实盘中需根据自己的策略及需求来改变值大小。更多关于买卖因子、选股策略等股票量化相关的知识，有兴趣的读者可以自行学习研究。

```
# 初始化资金 200 万元
read_cash = 2000000
# 每笔交易的买入基数资金设置为万分之 15
abupy.beta.atr.g_atr_pos_base = 0.0015
```

使用 `run_loop_back` 运行策略进行全市场回测：

```
from abupy import AbuFactorBuyBreak
from abupy import AbuFactorAtrNStop
from abupy import AbuFactorPreAtrNStop
from abupy import AbuFactorCloseAtrNStop
from abupy import abu
# 设置选股因子，None 为不使用选股因子
stock_pickers = None
# 买入因子使用向上突破因子
buy_factors = [{'xd': 60, 'class': AbuFactorBuyBreak},
               {'xd': 42, 'class': AbuFactorBuyBreak}]
# 卖出因子
sell_factors = [
    {'stop_loss_n': 1.0, 'stop_win_n': 3.0,
     'class': AbuFactorAtrNStop},
```

```

{'class': AbuFactorPreAtrNStop, 'pre_atr_n': 1.5},
{'class': AbuFactorCloseAtrNStop, 'close_atr_n': 1.5}
]
# 全市场
choice_symbols = None
# 使用 run_loop_back 运行策略, 5 年历史数据回测
abu_result_tuple, kl_pd_manger = abu.run_loop_back(read_cash,
                                                    buy_factors,
                                                    sell_factors,
                                                    stock_pickers,
                                                    choice_symbols=
                                                    choice_symbols,
                                                    n_folds=5)

```

下面为全市场中九成股票在过去 5 年的回测结果（切割训练集与测试集默认为 9:1），可以看到最终策略的收益没能跑赢大盘，如图 3-9 所示。股票市场的 2/8 原则体现在市场中是 80% 的股票会弱于大盘，只有 20% 的股票会强于大盘。从另一个角度来说，海龟交易中介介绍的是突破法作为趋势买入信号在实战中确实比较尴尬，因为它的信号太明显了。即便仅使用简单的双均线突破策略也都比这个突破策略效果更好，因为均线的参数设定有起点因素，使得信号不趋于同化（读者可自行测试）。

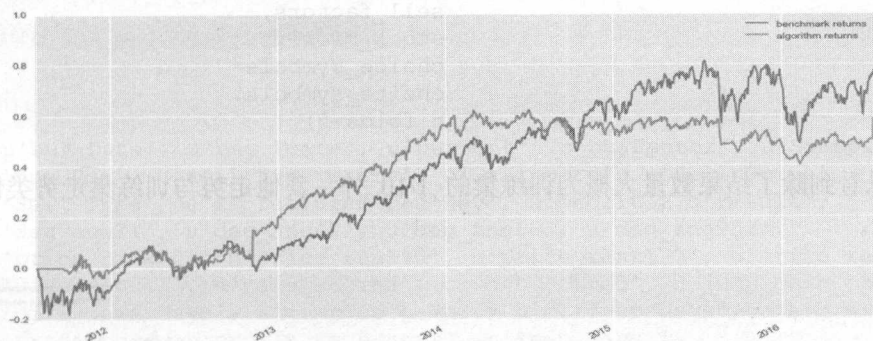


图 3-9 度量训练集全市场回测

```

from abupy import AbuMetricsBase
metrics = AbuMetricsBase(*abu_result_tuple)
metrics.fit_metrics()
metrics.plot_returns_cmp(only_show_returns=True)

```

输出：

```

买入后卖出的交易数量:80261
胜率:44.2%
平均获利期望:11.28%
平均亏损期望:-6.12%
盈亏比:1.2097

```



策略收益: 58.05%

基准收益: 77.87%

策略年化收益: 11.61%

基准年化收益: 15.57%

策略买入成交比例: 32.67%

策略资金利用率比例: 86.22%

策略共执行 1260 个交易日

下面设置 `g_enable_last_split_test`, 使用刚才切割股票池中的测试集, 它使用 `pickle` 读取之前已经切割好的本地化测试集股票代码序列, 初始资金依然为 200 万元, 但是提高 `g_atr_pos_base` 为 0.015 (因为切割训练集与测试集的比例是 9:1, 所以提高 `g_atr_pos_base` 为之前的 10 倍)。

```
abupy.env.g_enable_train_test_split = False
# 使用刚才切割股票池中的测试集 symbols
abupy.env.g_enable_last_split_test = True
read_cash = 2000000
abupy.beta.atr.g_atr_pos_base = 0.015
choice_symbols = None
abu_result_tuple_test, _ = abu.run_loop_back(read_cash,
                                              buy_factors,
                                              sell_factors,
                                              stock_pickers,
                                              choice_symbols=
                                              choice_symbols,
                                              n_folds=5)
```

可以看到除了结果数量大概为训练集的 1/10 外, 其他走势与训练集走势类似, 如图 3-10 所示。

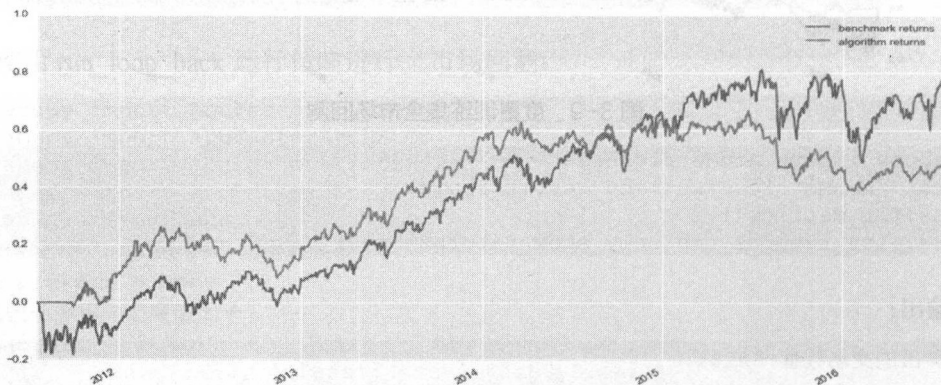


图 3-10 度量测试集全市场回测

```
metrics = AbuMetricsBase(*abu_result_tuple_test)
metrics.fit_metrics()
```



```
metrics.plot_returns_cmp(only_show_returns=True)
```

输出：

买入后卖出的交易数量:9381

胜率:43.58%

平均获利期望:9.91%

平均亏损期望:-6.68%

盈亏比:1.147

策略收益: 49.12%

基准收益: 77.87%

策略年化收益: 9.82%

基准年化收益: 15.57%

策略买入成交比例: 29.56%

策略资金利用率比例: 86.62%

策略共执行1260个交易日

上面是在所有生成结果的 `orders_pd` 中添加了交易买入信号发出时刻的机器学习特征元素（由于 `g_enable_ml_feature` 的设置），最终会输出如下所示的 `deg_ang21`、`price_rank90`、`jump_up_power`、`wave_score1`，等等。

```
abu_result_tuple.orders_pd.columns
```

输出：

```
Index(['u'buy Date', 'u'buy Price', 'u'buy Cnt', 'u'buyFactor', 'u'Symbol',
      'u'buy Pos', 'u'sell_type_extra', 'u'Sell Date', 'u'Sell Price',
      'u'Sell Type', 'u'ml_features', 'u'key', 'u'profit', 'u'result',
      'u'deg_ang21', 'u'deg_ang42', 'u'deg_ang60', 'u'deg_ang252',
      'u'price_rank60', 'u'price_rank90', 'u'price_rank120', 'u'price_rank252',
      'u'wave_score1', 'u'wave_score2', 'u'wave_score3', 'u'jump_down_power',
      'u'diff_down_days', 'u'jump_up_power', 'u'diff_up_days', 'u'atr_std',
      'u'profit_cg', 'u'profit_cg_hunder', 'u'keep_days'],
      dtype='object')
```

### 3.3.2 基于特征的交易预测

上面的 `deg_ang`、`price_rank`、`wave_score`、`ump_down_power` 等全部为买入时的特征，如果我们可以使用机器学习技术学习这些特征生成模型，在之后的交易中通过模型来预测交易是否可以盈利来提升策略的盈利能力，那么我们是不是就拥有了类似在童话世界中一样的预测能力了呢，下面开始尝试。

首先使用几个综合特征来训练模型，特征选择如下所示：

```
from abupy import AbuUmpMainMul
```

```
mul = AbuUmpMainMul.UmpMulFiter(orders_pd=abu_result_tuple.orders_pd,
                                scaler=False)
mul.df.head()
```

输出如表 3-5 所示。

表 3-5 回测结果

	result	deg_ang252	price_rank252	wave_score3	atr_std
2011-09-21	1	8.121	0.984	1.372	0.742
2011-09-21	1	-2.201	0.947	0.000	0.198
2011-09-21	0	40.096	1.000	1.277	0.046
2011-09-21	0	2.914	1.000	1.214	0.110
2011-09-21	1	4.210	1.000	1.339	0.890

上面输出的每一行实际上都代表一次交易回测，result 代表这次交易的最终结果（0：亏损，1：盈利），deng\_ang252 代表买入信号发生时刻向前 252 天的交易日收盘价格拟合曲线角度特征值，其他的列也都为买入时刻的股票特征值（后续章节会讲解 price\_rank252 等特征的含义）。下面将以 result 为 y，以其他特征为 x，以能预测涨跌作为目标，对特征进行训练、验证等机器学习操作。

UmpMulFiter 类型对象 mul 默认使用 SVM 作为分类器（一种分类模型，本书未介绍，读者可自行研究），使用 cross\_val\_accuracy\_score() 函数进行交叉验证，从输出结果来看，根本无法达到可预测能力。

```
mul().cross_val_accuracy_score()
```

输出：

```
accuracy mean: 0.507419337402
array([0.5146381, 0.51526099, 0.51482681, 0.50685273, 0.50996761,
       0.49538998, 0.5049838, 0.50149514, 0.50485921, 0.505919])
```

下面使用随机森林尝试，结果基本一样。

```
mul().estimator.random_forest_classifier()
mul().cross_val_accuracy_score()
```

输出：

```
accuracy mean: 0.521635563078
array([0.51501184, 0.53407251, 0.52803389, 0.5074757, 0.52367306,
       0.5251682, 0.520309, 0.52230252, 0.52093197, 0.51937695])
```

下面使用历史拟合角度特征来实验一下，使用 adaboost，特征输出如表 3-6 所示。

```

from abupy import AbuUmpMainDeg
deg = AbuUmpMainDeg.UmpDegFiter(orders_pd=abu_result_tuple.orders_pd)
# 分类器使用 adaboost
deg().estimator.adaboost_classifier()
deg.df.head()

```

输出如表 3-6 所示。

表 3-6 特征输出

	result	deg_ang21	deg_ang42	deg_ang60	deg_ang252
2011-09-21	1	1.442	3.342	-0.345	8.121
2011-09-21	1	4.477	-0.771	-2.201	-2.201
2011-09-21	0	10.349	9.229	11.483	40.096
2011-09-21	0	1.717	3.830	2.099	2.914
2011-09-21	1	2.748	2.377	-1.083	4.210

输入:

```
deg().cross_val_accuracy_score()
```

输出:

```

accuracy mean: 0.552472450867
array([0.55587393, 0.55687056, 0.55743833, 0.52840768, 0.55183155,
       0.55693995, 0.55631697, 0.55407426, 0.55532021, 0.55165109])

```

这个效果似乎不错，但别高兴得太早，看看混淆矩阵的分布，可以发现，它正确率高的原因是模型大多数的预测都是 0，典型的非均衡。

```
deg().train_test_split_xy()
```

输出:

```

(80261, 4)
(72234, 4)
(8027, 4)
accuracy = 0.57
precision_score = 0.56
recall_score = 0.01
      precision    recall  f1-score   support

0.0         0.57         0.99         0.72         4567
1.0         0.56         0.01         0.02         3460

avg / total         0.56         0.57         0.42         8027
Confusion Matrix [[4540   27]
 [3426  34]]
Predicted

```

	0	1
Actual	4540	27
	3426	34

下面使用更多的特征来实验一下，使用 `adaboost`，特征输出如下所示。

```
from abupy import AbuUmpMainFull
full = AbuUmpMainFull.UmpFullFiter(orders_pd=
                                     abu_result_tuple.orders_pd)
# 继续使用 adaboost
full().estimator.adaboost_classifier()
# 查看 full 所有特征名称
full.df.columns
```

输出：

```
Index([u'result', u'deg_ang21', u'deg_ang42', u'deg_ang60', u'deg_ang252',
       u'price_rank60', u'price_rank90', u'price_rank120', u'price_rank252',
       u'wave_score1', u'wave_score2', u'wave_score3', u'jump_down_power',
       u'diff_down_days', u'jump_up_power', u'diff_up_days', u'atr_std'],
      dtype='object')
```

`full` 进行交叉验证：

```
full().cross_val_accuracy_score()
```

输出：

```
accuracy mean: 0.533646282273
array([0.55051701, 0.54304223, 0.53688014, 0.50137055, 0.52068278,
       0.51968602, 0.5428607, 0.52878146, 0.54846748, 0.54417445])
```

可以看到交叉验证的准确率最高值依然不高，交叉验证都没有通过，由于篇幅限制，这里就不将这个模型带入我们切割的测试集中去做验证了。

### 3.3.3 破灭的童话——真实世界的机器学习

童话世界中的机器学习如此给力，为什么会在真实世界中败的体无完肤呢？

在混沌理论中，世界上任何事物都可以通过数字放大，从而影响其他不相关的事物。一只亚马逊雨林的蝴蝶翅膀振动，可能引起美国得克萨斯州的一场龙卷风，如图 3-11 所示。真实世界的因素总是无限多的，但机器学习技术只能在有限的特征中挖掘其内在表达式。这就是我们上面失败的根本原因！



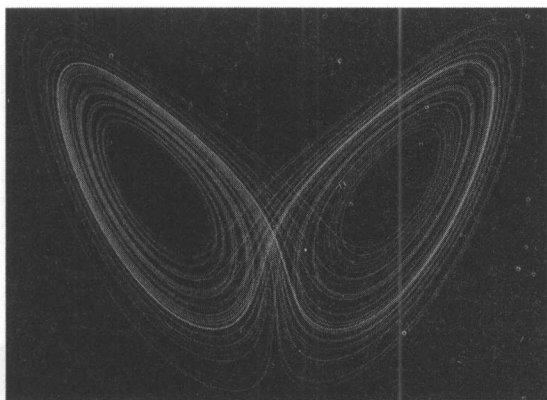


图 3-11 蝴蝶效应

阅读到这里你可能会问，机器学习技术对真实世界的问题一点帮助都没有吗？

当然不是！过去很多人已成功地将机器学习技术运用到各个任务中解决实际问题，因为在特定的任务内容范围下，影响事物的内在因素的数量表现出相对有限！在这种情况下，机器学习可以做到一定程度的有效预测——这就像某个数学几何理论，放到宇宙世界中看，这个理论是不成立的，但在小范围直观世界中却是可行的（感兴趣的读者请自行搜索“平行公设”相关内容）。

具体到股票领域中也是如此，真实市场的股价走势是由背后参与的交易者推动的，而交易者又在无限的因素影响下进行交易，所以直接拿机器学习预测并不靠谱。但是，通过历史数据可以发现，群体交易者行为中包含的非理性行为表现出一定的固定模式（由于人性的弱点）。所以在交易者参与下的股票市场中，存在一种介于预测和混沌之间的有限状态，这种状态可以使用概率来描述。虽然我们无法直接预测，但可以通过算法来找到这些概率分布，在交易时获得一些概率上的优势。

当我们把一种技术落地到特定领域时，它的设计思想才是关键，机器学习不是只能用来对股票的k线图进行识别分类，它的很多实现思想才是在股票量化交易上的闪光点，有些人已经在交易策略中合理地使用机器学习算法，并且运行收益良好。

具体如何将机器学习技术落地到股票量化，这是一个大话题，我在公众号 `abu_quant` 中会有更详细的介绍。本书篇幅有限，就不说题外之话了。



## 第二篇

# 深度学习篇

所有这些突破在本质上其实都是同一个突破，它们都是靠一组热门人工智能技术取得的，这种技术的名字叫作深度学习。

# 深度学习：背景和工具

最近，世界似乎变得有点快。

某一天，我发现以前看不懂的外文书籍可以通过 Google 翻译直接阅读了；iPhone 的 Siri 能和我进行简单的对话了；昨天上班突然不用再刷卡，开始对着 iPad 刷“脸”（人脸识别）了；百度、Google 都在玩自己制造的无人车原型；而亚马逊居然开始用无人机送货了。

本章，让我们先来熟悉深度学习的背景和相关类库。

## 4.1 背景

世界尽头的地方，是雄狮落泪的地方，是月亮升起的地方，是美梦诞生的地方。

——大卫《人工智能》

深度学习的起源，需从人工智能的历史背景开始说起。

### 4.1.1 人工智能——为机器赋予人的智能

人类对创造智能生命的渴望渊源已久，在很早的希腊神话中已经出现了机械人和人造人，中世纪则出现了使用巫术或炼金术将意识赋予无生命物质的传说。Samuel Butler 的《机器中的达尔文（Darwin among the Machines）》一文（1863）探讨了机器通过自然选择进化出智能的可能性。至今，人工智能（Artificial Intelligence）仍然是科幻小说、文艺作品中的重要元素。

### 4.1.2 图灵测试

1950 年，图灵发表了一篇划时代的论文，文中预言了创造出具有真正智能的机器的

可能性。这就是著名的图灵测试：如果一台机器能够与人类展开对话（通过电传设备）而不被辨别出其机器身份，那么就称这台机器具有智能，如图 4-1 所示。

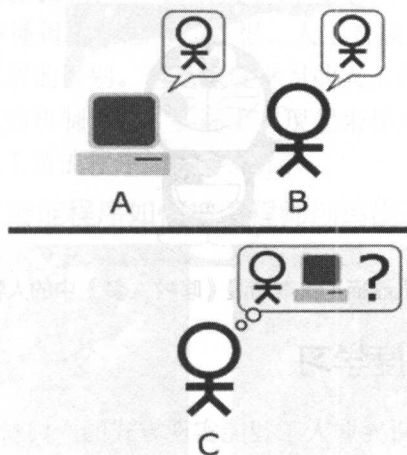


图 4-1 图灵测试（图片出自果壳网文“人工智能下一步，通过图灵测试”）

图灵测试能够令人信服地证明“思考的机器”是可能的。直至 2014 年 6 月 8 日，首次有计算机通过了图灵测试，尤金·古斯特曼在雷丁大学（University of Reading）所举办的测试成功骗过了研究人员，令他们以为“它”是一位名为 Eugene Goostman 的 13 岁男孩，但后来有文章指它其实并非真正地通过了测试。

图灵测试在一定程度上给出了划分智能与非智能的标准，从而拉开了人工智能史的大剧。那时的科学家曾一度脑洞大开，比方说在 1962 年，人工智能的先驱、经济学诺贝尔奖获得者，卡内基梅隆大学的西蒙教授就曾断言：“二十年内，计算机将完成人所能做到的一切工作。”

### 4.1.3 强人工智能 vs 弱人工智能

然而，半个世纪过去了，能够像人一样推理的机器依旧没有出现。“理想是丰满的，现实是骨感的”，在先哲的预言被不幸“打脸”后，科学家痛定思痛，将曾经的目标按实现方式分为两大领域：让机器真正的思考：强人工智能；让机器表现出智能的行动：弱人工智能。

弱人工智能并不是具备真正意义上的智能，而是利用计算机强大的运算能力和海量的历史数据，“伪装”出智能的行为。比如在图灵测试中，开发人员如果预先猜到了测试人员对机器所有可能的问题，并且将人工的答案编入程序中，那么测试人员就完全区别不

出对面是人还是机器，这就是一种弱人工智能。而强人工智能，则是真正的像人类那样思考和决策，目前的典型例子都是在小说、电影等文艺作品里，比如机器猫，如图 4-2 所示。



图 4-2 机器猫（日本动漫《哆啦 A 梦》中的人物形象）

#### 4.1.4 机器学习和深度学习

机器学习和深度学习是研究弱人工智能实现方式的一门科学。

类似第一章介绍的浅层学习模型，训练好的模型应用于未知的新数据时，可以表现出一些类似智能的预测能力。让计算机通过拟合数据，“伪装出”非凡的智能行为便是机器学习的终极目标。

深度学习是机器学习的一个分支，深度学习模型的学术别名是“人工神经网络”。深度学习同样是研究实现弱人工智能的科学。和传统机器学习模型相比较，深度学习模型除了依赖统计学之外，还借鉴了很多生物学的知识。

深度学习模型在提出后，一开始并没有多少人理会，基于两个原因：

- (1) 训练神经网络需要海量的数据，而当时互联网并不成熟，数据储备不足。
- (2) 良好的神经网络模型需要大量的神经元计算（比如狒狒大脑拥有 860 亿个神经元），庞大的计算量是当时的硬件无法胜任的。

但是随着摩尔定律的支撑，计算性能每年都有指数级的提高，同时互联网信息也迅速丰满起来。有了海量的计算能力以及庞大的数据仓库，VC 机构 A16Z 的合伙人 Frank Chen 说：“这就是深度学习的寒武纪大爆发”。前百度首席科学家吴恩达也说：“在过去，许多标普 500 强的 CEO 希望自己能早点意识到互联网战略的重要性。我想从现在开始的今后 5 年，也会有一些标普 500 强的 CEO 后悔没有早点思考自己的 AI 战略。”

#### 4.1.5 过度的幻想

在互联网浪潮之下，又有不少开始过分地幻想这一机器学习技术，甚至炮制出“人

工智能的发展可能意味着人类的灭亡”等人工智能威胁论。

确实，神经网络擅长模式识别——在某些小领域有时候表现得比人类自身还好。但它们还是弱人工智能，并不懂得主动思考、联想。人工智能可以使我们了解“智能”，而人脑是具备智能的，这是本质的区别。包括深度学习的人工神经网络、进化算法、遗传算法等，这些名字听起来和人脑机制一样，实际上实现起来和人脑思维的方式还是有着非常大的差别，不同人眼中的人工智能程序如图 4-3 所示。

人工智能程序如何被工程师创造出来的？

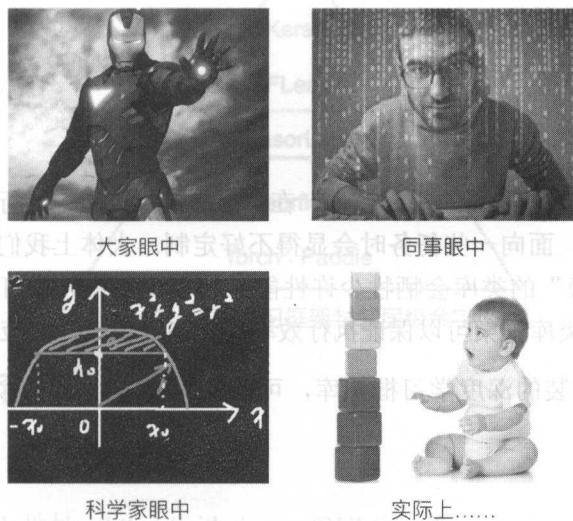


图 4-3 不同人眼中的人工智能程序

### 4.1.6 回顾

- 机器学习是弱人工智能的实现。
- 深度学习是机器学习的一个分支，借鉴了生物学知识设计学习模型。

## 4.2 深度学习框架简介

工欲善其事，必先利其器。

——孔子《论语·卫灵公》

本节评测几个深度学习实现框架。



## 4.2.1 评测方式

这几年深度学习的框架类库呈井喷状，并且很多框架均已开源。开源类库之间竞争激烈，排名不断变化，本节给出的评分在到你手中时可能已经不再适用。我们从以下几个方面给出一个简单的评测。

### 1. 支持的语言

目前主流的套路是底层用 C 类语言实现计算，上层给开发者提供 Python 接口。当然也有类库支持别系语言的框架实现。

### 2. 封装等级

封装等级不完全等于易用性。

封装越高层的类库需要写的代码越少，在如 Keras 等类库中几行代码就能完成一个不错的模型；但对应的，面向一些任务时会显得不好定制。大体上我们按实验级、工业级评估封装层次。“实验级”的类库会牺牲少许性能，用极少的代码即可快速建模看到模型效果。而“工业级”的类库框架可以保证执行效率和定制自由度，对应的有一定上手难度。

另外，一些高封装的深度学习框架库，可能依赖于其他一些底层的框架库作为后端执行引擎。

### 3. 执行效率

“这一评估其实并不是最重要的，除非你真的很在乎速度。一般可能更关注“封装等级”和“关注程度”，深度学习类库都在快速地改版中，每一版本的执行速度可能差异很大。基本上高封装的类库必然会慢一些，而更多人关注的类库必然会有更快的进化速度。

### 4. 可定制程度

模块的可定制能力对于资深人群而言额外重要。

### 5. 文档友好度

对于深度学习的新手，这一点很重要。尤其英语不好的朋友会格外在意中文文档的支持度。

### 6. 人气

业界的关注程度能反映这个类库上升的潜力，这一点在目前百家争鸣的格局下额外重要。

## 4.2.2 评测对象

我们主要关注以下几个开源的深度学习库的对比：Caffe、TensorFlow、Keras、TFlearn。其他如 MXnet、Theano、Torch（Facebook 力推的深度学习框架，主要开发语言是 C 和 Lua）、Paddle 等由于开发语言、人气等原因并没有列入比较范围。

先给出这些类库整体的封装层级直观图，如图 4-4 所示。

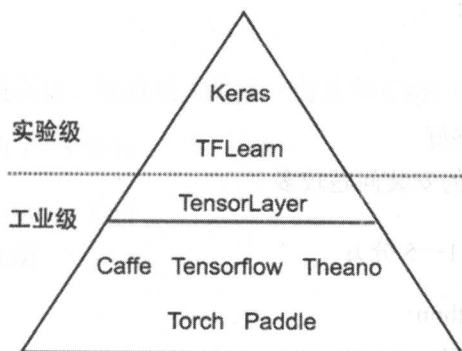


图 4-4 深度学习框架封装层级金字塔

## 4.2.3 深度学习框架评测

### 1. Caffe

- 项目 GitHub 地址：<https://github.com/BVLC/caffe>
- 中文文档：有一份中文翻译的官方教程 <http://caffe.cn/?/page/tutorial>
- 英文文档：<http://caffe.berkeleyvision.org/>
- 社区：<http://www.caffecn.cn/>

Caffe 是一个主流的工业级深度学习工具。它开始于 2013 年底，由 UC Berkely 的 Yangqing Jia（贾扬清）老师编写和维护，具有出色的卷积神经。

Caffe 的特色是支持 Shell 脚本+配置文件的使用方式。由于 Shell 函数大多都是极度高效的内核函数，所以 Caffe 的运行速度非常快，特别适合深层 CNN 任务。Caffe 的最大问题是各种安装环境问题比较多，对新人不够友好，而且对一些模型（如 RNN）支持力度不足。关于 CNN、RNN 后续章节会介绍。

值得一提的是，在本书编写期间，Facebook 开源了应用在自家平台上的深度学习框架：Caffe2，优化了移动设备部署和大规模机器部署，官方甚至提供了从 Caffe 到 Caffe2 迁移的教程，感兴趣的读者可以自行研究。

优点:

- 快
- 模块化好
- 支持命令行模式使用
- 支持分布式
- 对 CNN 支持很好

缺点:

- 对 RNN 支持不够好
- 不同环境下反映的安装问题较多

我们的简评 (数值项 1—5 分):

- 支持的语言: Python
- 封装等级: 工业级
- 执行效率: 5
- 定制能力: 4.8
- 文档友好度: 4.5
- 人气: 4.8

## 2. TensorFlow

- 项目 GitHub 地址: <https://github.com/tensorflow/tensorflow>
- 中文文档: [http://www.tensorfly.cn/tfdoc/how\\_tos/overview.html](http://www.tensorfly.cn/tfdoc/how_tos/overview.html)
- 社区: <http://www.tensorfly.cn/>

TensorFlow 是 Google 开源的其第二代深度学习技术, 被使用在 Google 搜索、图像识别以及邮箱的深度学习框架。TensorFlow 采用数据流图 (data flow graphs), 以 graph 为计算框架, 以 session 为运行环境, 以 operation 为运算单元。同时可以用 TensorBoard 来展现你的 TensorFlow 的模型图像, 绘制图像生成的定量指标图以及附加数据。

TensorFlow 支持分布式可移动设备, 支持 C++、Python 开发模型, 可以说是目前最火的开源库。TensorFlow 的特色是人气高、生态系统好。现在有很多基于 TensorFlow 二次封装的实验级类库, 帮助 TensorFlow 形成生态系统。和 Caffe 相比, TensorFlow 云集的大牛资源多, 所以推进快, 反馈的安装环境等阻碍推广的问题不多。

优点：

- 人气高，资源充分，生态系统好
- 可移植性好
- 安装问题少
- 对模型调试的可视化支持好

缺点：

TensorFlow 的优点是灵活，缺点是太灵活。而且和 Caffe 相比，速度慢。

我们的简评（数值项 1—5 分）：

- 支持的语言：Python、C++
- 封装等级：工业级
- 执行效率：4.8
- 定制能力：5
- 文档友好度：5
- 人气：5

### 3. Keras

- 项目 GitHub 地址：<https://github.com/fchollet/keras>
- 中文文档：<https://keras-cn.readthedocs.io/en/latest/>

Keras 由纯 Python 编写而成，是基于 Theano 或 TensorFlow 之上的二次封装类库。也就是说，使用 Keras 需要先预装 Theano 或 TensorFlow 两者之一（推荐 TensorFlow）。Keras 为快速实验而生，如果说 Caffe 是“极速主义”，那么 Keras 就是“极简主义”，Keras 的模块设计极为简洁，很适合源码阅读学习。

总之，Keras 的特色是简单，无论是使用还是学习源码，目前它都是新人的不二选择。

优点：

- 简单即优雅
- 适合源码学习

缺点：

Keras 的缺点是极简主义带来的问题——很难定制。为了极简，Keras 遮蔽了很多可

调式的选项，也遮蔽了很多实现细节，所以让使用者感觉额外简单，但一些复杂的神经网络模块用 Keras 很不好实现。Keras 可以说是学术界的宠儿，却对工业界的朋友不怎么友好。Keras 目前的运行效率也不太够，社区里有人提到官方正在努力改善这些问题，喜欢 Keras 的朋友可以期待一下。

我们的简评（数值项 1—5 分）：

- 支持的语言：Python
- 封装等级：实验级；
- 执行后端：Tensorflow or Theano
- 执行效率：3.5
- 定制能力：4
- 文档友好度：5
- 人气：4.8

#### 4. TFLearn

- 项目 GitHub 地址：<https://github.com/tflearn/tflearn>
- 中文文档：暂无
- 英文文档：[http://tflearn.org/getting\\_started/](http://tflearn.org/getting_started/)

TFLearn 在写法和其他很多方面都和 Keras 类似，基于 TensorFlow，也是由纯 Python 编写而成，封装程度为“实验级”。TFLearn 的模型抽象是 functional model，和 Keras 对比，TFLearn 的优点是

- Keras 在 Theano（业内曾经的明星）时代就推出了，目前 Keras 还在兼顾 Theano。而 TFLearn 一开始就直接抱了 TensorFlow 的大腿，目前看 TFLearn 在模块设计上更干净。
- 比起 Keras 的极简主义，TFLearn 的 API 设计给了使用者更大的定制空间，执行效率也比 Keras 略好一点。可以说，TFLearn 是比 Keras 更靠近“工业级”的设计。
- 支持强化学习。

缺点：

中文文档支持不好。这一点对英语不好的朋友而言很致命。人气也有点不足，未来未知。我们的简评（数值项 1—5 分）：



- 支持的语言：Python
- 封装等级：实验级；执行后端：TensorFlow
- 执行效率：3.8
- 定制能力：4.2
- 文档友好度：4
- 人气：4

#### 4.2.4 小结

虽然深度学习框架的现状是群雄争霸，不过这些框架的直观使用感受其实非常相似。极端选择中，极速主义者推荐 Caffe，极简主义者推荐 Keras，入门学习推荐 Keras，高端玩家选自己拿手的就好。

本书中，对于原理性实验主要采用 Keras 实现，对于工程实践主要采用 Caffe 实现。

### 4.3 深度学习框架快速上手

在熟悉一些基本的概念以后，大部分深度学习框架上手都很容易。本书主要使用 Keras 和 Caffe，安装教程见附录 B。

#### 4.3.1 符号主义

大部分深度学习库都是以符号主义的方式使用。

符号主义是说在建立模型任务时，首先定义各种变量，建立一个整体的计算图。计算图规定了各个变量之间的计算关系。建立好的计算图需要编译已确定其内部细节，但此时的计算图还是一个“空壳”，里面没有任何实际的数据，只有当你把需要运算的数据放进去后，才能在整个模型中形成数据流，从而形成输出值。

这样设计的原因是，深度学习任务中常常要并行处理海量的数据（100GB 以上），在多层模型之间进行复杂的计算。我们希望计算任务能够高效快速地实现，如一些底层高效地编程语言，同时，希望上层的使用者能够简单地进行开发（如用 Python 定义模型），尤其是那些希望快速实验的学术人群。最后，还希望数据尽可能地少迁移。在底层语言和上层语言环境切换时，必然会带来不少损耗，尤其是数据 I/O 方面。因此，大多数符号主义的类库首先会用诸如 Python 代码构建符号图，整体安排好图的哪一部分应该被运行。

好比我们告诉厨房（深度学习框架），要做一道菜（设计某个模型）。“Python”是个

好传话筒，虽然不怎么烧菜，但好沟通，它负责做好菜谱（描述符号计算图）传递给下游厨房。深度学习引擎好比厨房小弟，为菜谱的各道工序调度做准备（模型编译、运行环境、数据流等）；“C 类语言”师傅的烧菜速度很快，但真的很难沟通，我们只让他在厨房小弟准备妥当当时照着菜谱干活就行，如图 4-5 所示。

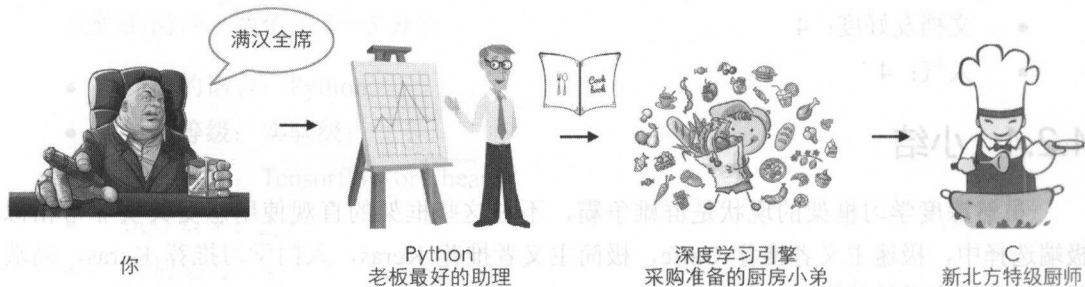


图 4-5 深度学习框架角色图

这样每个角色各尽所能。因为每次菜谱改动，都会让跑腿的厨房小弟跑一趟市场/仓库获取原材料，所以我们希望菜谱先一次定义好全部工序，让厨房小弟少跑路——全局层面上高效地“做菜”，这是所有符号主义设计的初衷。

### 基础数据结构

要定义好整个“菜谱”，就需要定义好每道工序的输入和输出。在实验级封装的深度学习库（比如 Keras），它们会遮蔽每个工序的输入输出，不会暴露给开发者；但其他深度学习库一般都有自己的数据结构保存这些输入输出值，并且暴露给使用方。比如 Caffe 中的 blob。这些数据结构都可以看作多维数组，沿着向量、矩阵的自然推广，比如：

- 0 阶，数值
- 1 阶，向量
- 2 阶，矩阵，二维数组（如灰度图片）
- 3 阶，矩阵序列，三维数组（如 RGB 彩图，灰度图数组等）
- 4 阶，blob，四维数组（如彩图数组，视频等）

.....

OK，需要知道的基础概念就这么多，接下来，让我们以一个图片识别的例子，上手深度学习库。

### 4.3.2 MNIST

我们将通过 MNIST 数据集上手深度学习库。完整代码、数据见本书随书示例代码。

MNIST 可以说是图像识别领域的“Hello World!”。它是 Google 实验室的 Corinna Cortes 和纽约大学柯朗研究所的 Yann LeCun 联合创建的一个手写数字数据库，每个样本数据是一张  $28 \times 28$  像素的灰度手写数字图片，每张图片对应一个数字，训练库有 60000 张手写数字图像，测试库有 10000 张。MNIST 数据集如图 4-6 所示。

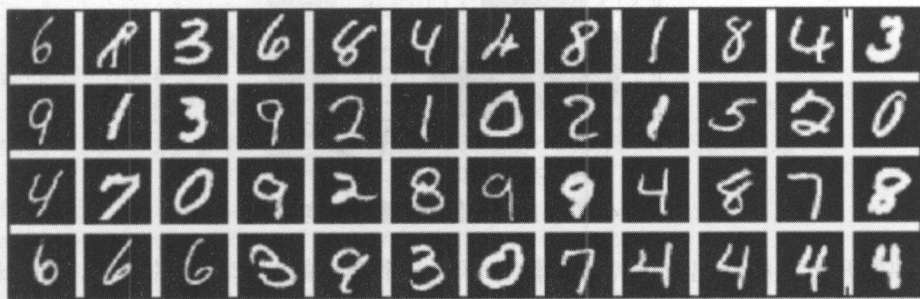


图 4-6 MNIST 数据集

可以看到，有少数图片扭曲得很厉害。我们将使用深度学习框架，在 MNIST 数据集上完成逻辑分类任务。

## 描述任务

问题：

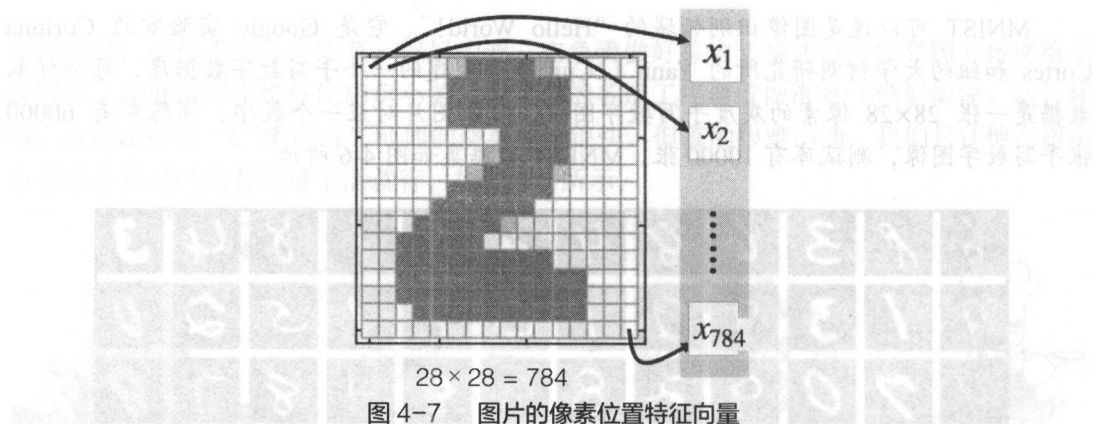
- 60 000 个训练样本，10 000 个测试样本，每个样本是一张  $28 \times 28$  的灰度图片。
- 0-9，共 10 个数字类别。

目标：训练模型，识别图片中的数字。

## 2. 构造样本向量

计算机将图片的光学转成二维矩阵存储，每个光学像素点可以看成是一个 0-255 的数值（对于彩色图片则是 RGB 三个色彩通道的数值的叠加）。首先，我们将这些像素数值先归一化到  $[0, 1]$  之间，然后展开成一个向量，长度是  $28 \times 28 = 784$ ，这就是一个样本向量。和之前例子中的样本向量做一下对比，思考一下现在的“特征”是什么？

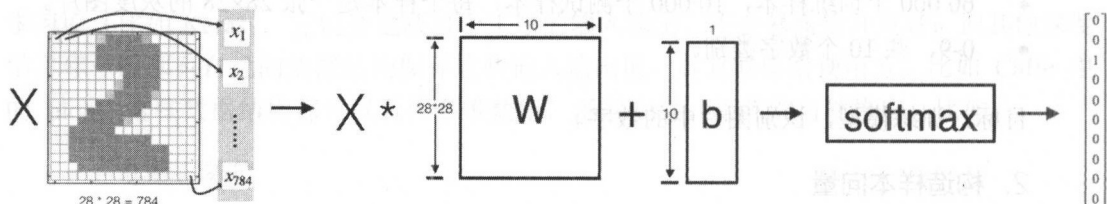
之前的例子中，在每个样本向量的特征位置上都有一个对应的特征意义，比如花瓣的长度、年龄等。这个图片向量也不例外，我们将图片按位置展开向量，相当于构造样本的“像素位置”特征，向量的每个特征位置会对应一个像素位置，如图 4-7 所示。



这种展开方式构造出来的特征显然并不是一个合适的解读样本的特征，同时展开图片的数字数组会丢失一些图片的二维结构信息，这些都让分类模型辨识样本变得困难，在第 5 章的 CNN 模型中你会看到一些设计独特的神经元模型如何解读构造更加合理的特征向量。

### 4.3.3 Keras 完成逻辑分类

下面先使用 Keras 完成训练目标，回顾逻辑分类的计算步骤：线性函数  $\rightarrow$  概率  $\rightarrow$  分类，如图 4-8 所示。



**备注：**

本书所有示例代码均可在 <https://github.com/bbfamily/abu> 中下载并使用，如下载地址有变动，请关注公众号:abu\_quant 获取最新地址。

机器学习环境部署见附录 A，深度学习环境部署见附件附录 B，随书示例代码运行环境部署见附录 C。

首先，生成图片样本向量。

加载 mnist 数据：

```

from __future__ import print_function
import numpy as np
np.random.seed(1337)
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import np_utils

batch_size = 128 # 梯度下降一个批 (batch) 的数据量
nb_classes = 10 # 类别
nb_epoch = 10 # 梯度下降 epoch 循环训练次数, 每次循环包含全部训练样本
img_size = 28 * 28 # 输入图片大小, 由于是灰度图, 因此只有一个颜色通道

# 加载数据, 已执行 shuffle-split (训练-测试集随机分割)
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# 以 TensorFlow 为后端, 归一化输入数据, 生成图片向量
X_train = X_train.reshape(y_train.shape[0], img_size).astype(
    'float32') / 255
X_test = X_test.reshape(y_test.shape[0], img_size).astype(
    'float32') / 255

X_train.shape, X_test.shape

```

输出:

```
((60000, 784), (10000, 784))
```

同时需要将每个样本标签 One-Hot 编码, 例如, 将 [5] 编码成  $\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ 。

```

# One-Hot 编码标签, 例如, 将 [3, 2, ...] 编码成 [[0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
[0, 0, 1, 0, 0, 0, 0, 0, 0, 0], ...]
Y_train = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)

```

接着创建模型。之所以能使用深度学习库完成逻辑分类, 是因为和一层全连接的神经网络其实是等价的, 在第 5 章 DNN 神经模型中我们会对原理进行更详细的讲解, 这里不需要深究。

在 Keras 中, 模型的容器抽象分两种: Model 和 Sequential。Model 与 TensorFlow 风格相似, Sequential 与 Theano 和 Caffe 相似。下面使用 Sequential 模型容器。

模型创建:

```

# 创建模型, 逻辑分类相当于一层全连接的神经网络 (Dense 是 Keras 中定义的 DNN 模型)
model = Sequential([

```



```
Dense(10, input_shape=(img_size,), activation='softmax'),
])
```

创建好的模型需要编译，就好像定义好的“菜谱”需要递给主厨优化工序一样。编译的选项参数中，“优化器”是指训练模型时使用哪种方式优化模型参数，这些优化算法基本都是各种变种的梯度下降算法，这里选择 SGD（随机梯度下降）。“loss”是指选择哪种损失函数，我们选择（多类别）交叉熵。

模型编译：

```
# 编译模型；rmsprop 是改进的梯度下降，暂时不需要了解
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

模型训练函数：fit 函数通过 samples\_per\_epoch 和 nb\_epoch 两个参数来确定什么时候训练终止，这两个参数指定了每个 epoch 包含多少个样本以及要训练多少个 epoch。

训练模型：

```
model.fit(X_train, Y_train,
         batch_size=batch_size, nb_epoch=nb_epoch,
         verbose=1, validation_data=(X_test, Y_test))
```

输出：

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [=====] - 2s - loss: 0.6040 - acc:
0.8479 - val_loss: 0.3408 - val_acc: 0.9080
Epoch 2/10
60000/60000 [=====] - 1s - loss: 0.3304 - acc:
0.9086 - val_loss: 0.3001 - val_acc: 0.9157
Epoch 3/10
60000/60000 [=====] - 1s - loss: 0.3014 - acc:
0.9163 - val_loss: 0.2852 - val_acc: 0.9197
Epoch 4/10
60000/60000 [=====] - 1s - loss: 0.2881 - acc:
0.9202 - val_loss: 0.2778 - val_acc: 0.9242
Epoch 5/10
60000/60000 [=====] - 1s - loss: 0.2800 - acc:
0.9220 - val_loss: 0.2723 - val_acc: 0.9238
Epoch 6/10
60000/60000 [=====] - 1s - loss: 0.2745 - acc:
0.9238 - val_loss: 0.2732 - val_acc: 0.9243
Epoch 7/10
60000/60000 [=====] - 1s - loss: 0.2703 - acc:
0.9249 - val_loss: 0.2741 - val_acc: 0.9248
Epoch 8/10
```

```
60000/60000 [=====] - 1s - loss: 0.2671 - acc:
0.9264 - val_loss: 0.2692 - val_acc: 0.9257
Epoch 9/10
60000/60000 [=====] - 1s - loss: 0.2646 - acc:
0.9271 - val_loss: 0.2700 - val_acc: 0.9260
Epoch 10/10
60000/60000 [=====] - 2s - loss: 0.2627 - acc:
0.9281 - val_loss: 0.2681 - val_acc: 0.9264
```

评估模型成绩：

```
# 测试
score = model.evaluate(X_test, Y_test, verbose=0)
print('accuracy: {}'.format(score[1]))
```

输出：

```
accuracy: 0.9264
```

## Keras 小结

Keras 的使用整体分为以下几步：

- (1) 准备 train-test 数据。
- (2) 定义模型。
- (3) compile 模型。
- (4) 给模型灌入数据流 I/O。

## 4.3.4 回顾

- 深度学习框架的符号主义。
- 按图片的“像素位置”特征构造样本向量。
- 使用 Keras 实现逻辑分类模型。

## 4.4 Caffe 实现逻辑分类模型

本节使用 Caffe 实现逻辑分类模型。Caffe 配置起来要比 Keras 复杂得多，但对应的能给你更大的定制空间和更高效的运行速度，尤其对于图片识别任务，Caffe 优化得很好。

本书只介绍本书用到的 Caffe 配置，更多配置请查阅官方文档。新手可以先跳过本节，仅阅读本书 Keras 相关的部分，在达到一定程度之后再选择 Caffe 作为进阶的工具。

## 4.4.1 Caffe 训练 MNIST 概览

现在我们先全程浏览一下如何使用 Caffe 训练一个数据集，后面再逐个详细介绍。

说明：

- /root/maxmon/lr 是工程目录。
- /root/caffe 是 Caffe 的安装目录。
- MNIST 数据集解压缩在 /root/maxmon/lr/data 下。

第一步：在终端运行 `create_mnist.sh` 脚本，生成数据原料 `train_lmdb` 和 `test_lmdb`，脚本代码如下：

```
# 将$DATA文件下的数据文件转换成$OUTPUT下的train_lmdb和test_lmdb

# 工程目录
PROJECT_ENV=/root/maxmon/lr
# Caffe 安装目录
CAFFE_ENV=/root/caffe

OUTPUT=$PROJECT_ENV/gen
DATA=$PROJECT_ENV/data
# 使用$CAFFE_ENV/build/examples/mnist/convert_mnist_data完成lmdb转换
BIN=$CAFFE_ENV/build/examples/mnist/convert_mnist_data

BACKEND="lmdb"

echo "Creating ${BACKEND}..."

mkdir -p $OUTPUT

$BIN $DATA/train-images-idx3-ubyte $DATA/train-labels-idx1-ubyte \
    $OUTPUT/train_lmdb --backend=${BACKEND}
$BIN $DATA/t10k-images-idx3-ubyte $DATA/t10k-labels-idx1-ubyte \
    $OUTPUT/test_lmdb --backend=${BACKEND}
```

注意，这里使用 Caffe 的 `example` 目录下的 `convert_mnist_data` 工具。Caffe 使用的编译好的工具都放在“/你的 Caffe 目录/build/tools/”目录下，建议把它放到环境变量 `PATH` 中，这样在终端操作时更加方便。你可以使用 `tools` 里的 `convert_imageset` 完成上面的转换工作。

第二步，将下面的内容复制到模型文件 `/root/maxmon/lr/pb/train_val.prototxt` 中：

```
name: "LogistRegression"
layer {
  name: "data" # 数据层
```



```

type: "Data"
top: "data"
top: "label"
include {
    phase: TRAIN # 标识，这个数据层用于训练集
}
transform_param {
    scale: 0.00392156862745 # 归一化缩小
}
data_param {
    source: "/root/maxmon/lr/gen/train_lmdb" # 训练集地址
    batch_size: 64 # batch 样本数目
    backend: LMDB
}
}
layer {
    name: "data"
    type: "Data"
    top: "data"
    top: "label"
    include {
        phase: TEST # 测试数据层
    }
    transform_param {
        scale: 0.00392156862745 # 归一化缩小
    }
    data_param {
        source: "/root/maxmon/lr/gen/test_lmdb" # 测试集地址
        batch_size: 64 # batch 样本数目
        backend: LMDB
    }
}
}
layer {
    name: "ip1" # DNN 输出层
    type: "InnerProduct"
    bottom: "data"
    top: "ip1"
    inner_product_param {
        num_output: 10 # 输出 10 个类别
        weight_filler {
            type: "xavier" # weight 初始化方式
        }
    }
}
}
layer {
    name: "accuracy" # 准确率计算层
    type: "Accuracy"
    bottom: "ip1"
    bottom: "label"
}

```

```

    top: "accuracy"
  }
  layer {
    name: "loss" # softmax+loss 损失函数计算值
    type: "SoftmaxWithLoss"
    bottom: "ip1"
    bottom: "label"
    top: "loss"
  }

```

第三步，将下面的内容复制到/root/maxmon/lr/pb/solver.prototxt 中：

```

net: "/root/maxmon/lr/pb/train_val.prototxt"
test_iter: 250 # 一次测试 250 个 batch
test_interval: 1000 # 每训练 1000 个 iter 测试一次
base_lr: 0.01 # 梯度下降至基础步长
display: 1000 # 每 1000 迭代显示一次当前的成绩
max_iter: 10000 # 最大多少个 iter
lr_policy: "step" # 梯度下降学习速率改变方式
gamma: 0.1 # 配合 lr_policy: "step" 的参数
momentum: 0.9 # 梯度下降学习速率动量因子
weight_decay: 0.0005 # 权重衰减因子
stepsize: 5000 # 配合 lr_policy: "step" 的参数
snapshot: 10000 # 每 10000 次迭代保存训练结果
snapshot_prefix: "/root/maxmon/lr/gen/snapshot_train" # 训练结果保存地址
solver_mode: CPU

```

最后，终端输入 “/root/caffe/build/tools/caffe train -solver /root/maxmon/lr/pb/solver.prototxt”，开始训练模型。Caffe 是边训练边测试的，iteration 10000 之后，模型在测试集的准确率在 91%~92% 左右，训练好的权重保存在 /root/maxmon/lr/gen/snapshot\_train\_iter\_10000.caffemodel 中。

## 4.4.2 Caffe 简介

Caffe 的使用特色是 Shell 脚本+配置文件的风格。每个训练步骤都是一个 prototxt 配置文件，使用 Shell 命令运行配置文件。这样的好处是 Caffe 的模块化非常出色，当你使用新模型，或者调整一些训练步骤时，只需改动对应的配置文件就可以了。

由于命令行内置的都是内核函数，本身速度极快，加上 Caffe 本身引擎速度也很不错，所以使用起来整体的速度不错。但 Caffe 的缺陷是反馈的各系统下安装问题较多，Caffe 的使用风格不太适合不熟悉 Shell 命令的程序员。

如图 4-9 所示，Caffe 模型的训练过程分为三部分。



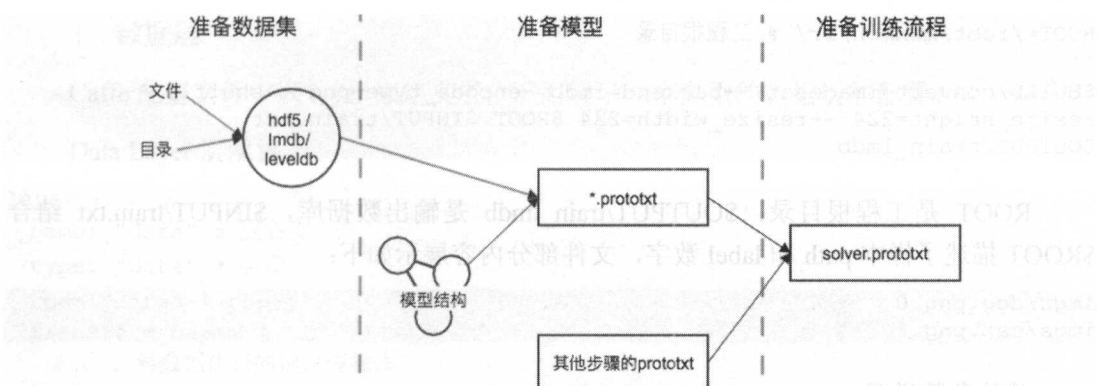


图 4-9 Caffe 训练流程

### 4.4.3 准备数据集

使用 Caffe 训练时，需要将数据集转换成 Caffe 支持的格式。Caffe 支持以下三种数据库类型：

- hdfs
- lmdb
- leveldb

#### 1. hdfs

支持并行 I/O、线程和其他一些现代系统和应用要求，轻量级数据库。

#### 2. lmdb

轻量级 Key-Value 数据库，不支持 SQL 语句。

#### 3. leveldb

Google 实现的高性能 Key-Value 数据库，单进程服务，不支持 SQL 语句，对海量数据存储支持比前两者略好。

Caffe 编译好的转换数据库的工具在“/你的 Caffe 安装目录/build/tools”目录下，比如对应一个图片文件夹/root/maxmon/project/imgs，可以用下面的脚本将文件夹下的 png 图片转换成图片大小为 224×224 的 lmdb：

```
INPUT=/root/maxmon/lr
OUTPUT=/root/maxmon/lr/gen
BUILD=/root/caffe # 我的 Caffe 目录
```

```
ROOT=/root/maxmon/1r/ # 工程根目录
```

```
$BUILD/convert_imageset --backend=lmdb --encode_type=png --shuffle --
resize_height=224 --resize_width=224 $ROOT $INPUT/train.txt
$OUTPUT/train_lmdb
```

*ROOT* 是工程根目录, *\$OUTPUT/train\_lmdb* 是输出数据库, *\$INPUT/train.txt* 结合 *\$ROOT* 描述了样本 path 和 label 数字, 文件部分内容展示如下:

```
imgs/dog.png 0
imgs/cat.png 1
```

其他参数说明。

- backend: 转换引擎, 支持 lmdb (默认)、hdf5、leveldb。
- encode\_type: 图片格式, 如 png、jpg、jpeg (默认)。
- shuffle: 随机化数据集。
- resize\_height、resize\_width: 输出图片尺寸。

补充两点:

- (1) 对于图片样本, 建议转换 Caffe 数据类型前先统一图片格式。
- (2) 本书后面的例子都使用 lmdb, 你可以自行探索其他数据库类型的使用。

#### 4.4.4 准备模型

深度学习框架都是以层的方式部署模型结构, 关于模型层具体会在后续讲解 (参见 5.3 神经元的深层网络结构)。正如上面例子中的模型文件 *train\_val.prototxt* 所示, 在 Caffe 中, 逻辑分类由如图 4-10 所示的三层构成, 层与层之间首尾相接 (top-bottom)。

- 数据层
- 全连接层
- 概率转换层

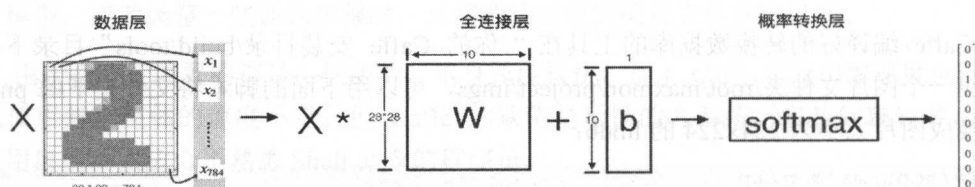


图 4-10 逻辑分类 MNIST 样本

## 1. 数据层

Caffe 配置文件中以 layer 来定义层，下面展示一个典型的数据层的配置。

Data Layer 层配置。

```
layer {
  name: "data" # 层名称
  type: "Data" # 类型
  top: "data" # top 层
  transform_param {
    # ... 对数据执行的预处理操作
  }
  data_param {
    source: "/root/maxmon/lr/gen/train_lmdb" # 训练集路径
    # ... 层参数
  }
  include {
    phase: TRAIN # TRAIN 或 TEST
  }
}
```

Caffe 是可以边训练边测试的。你可以划分训练—测试集之后，构建一个包含 phase: TRAIN 的 layer 告诉 Caffe 这是一个训练用的数据层，下面再建立一个包含 phase: TEST 的 layer，然后将这些配置都放在模型文件中。

还有一些其他数据层类型。

- type: MemoryData 内存数据。
- type: HDF5Data hdf5 数据。
- type: ImagesData 图片数据。

## 2. DNN 全连接层

在深度学习网络结构中，逻辑分类可以视为一层全连接的 DNN（DNN 见第 5 章深度学习模型）。InnerProduct Layer 是全连接层的内积层，用于描述矩阵乘积运算。

InnerProduct Layer 层配置。

```
layer {
  name: "ip1"
  type: "InnerProduct" # 注意类型为 InnerProduct
  bottom: "data" # 接在 name 为 data 的层之后
  top: "ip1"
  inner_product_param {
```

```

num_output: 10 # 这一层的输出维度
weight_filler {
  type: "xavier" # 训练权重初始化
}
}
}

```

Caffe 的层权重 `weight` 的初始化选择“xavier”就好。xavier 保持输入和输出的方差一致，避免了所有输出值都趋向于 0。

### 3. 几率转换层

模型最后输出的是“样本是某一类别的几率数值”，所以最后一层要接上几率转换层。二分类建议使用 Sigmoid 层，多分类建议使用 Softmax 层。

Softmax Layer 层配置。

```

layer {
  name: "loss"
  type: "Softmax" # 二分类用 Sigmoid
  bottom: "ip1"
  bottom: "label" # bottom 接输出 label
  top: "loss"
}

```

出于工程计算优化的目的，可以把 Softmax 计算和损失函数 loss 计算一起进行。

SoftmaxWithLoss Layer 层配置。

```

layer {
  name: "loss"
  type: "SoftmaxWithLoss" # 二分类用 SigmoidCrossEntropyLoss
  bottom: "ip1"
  bottom: "label"
  top: "loss"
}

```

深度学习框架中的每个操作都用“层”描述，下面定义计算准确率的操作层。

```

layer {
  name: "accuracy"
  type: "Accuracy"
  bottom: "ip1"
  bottom: "label"
  top: "accuracy"
}

```

## 4.4.5 模型训练流程

solver.prototxt 是 Caffe 中描述训练流程及调试参数的配置文件。

solver.prototxt 文件内容示意：

```
net: "/.../train_val.prototxt" # 训练用的模型结构配置 path
# .....其他模型训练参数，后续会讲解
display: 1000 每1000个梯度 iteration 测试一次当前的成绩
snapshot: 10000 # 每10000个梯度 iteration 保存一次模型当前的权重
snapshot_prefix: "/.../snapshot_train" # 保存模型权重的路径
solver_mode: CPU # CPU 模式还是 GPU 模型
```

要想理解其他的参数，首先需要掌握随机梯度的知识，在本书“4.5 solver Caffe 实现 MLP-调试参数”中会详细介绍，这里暂且不需要了解。

可以在终端运行“/你的 Caffe 目录/build/tools/caffe train -solver solver.prototxt”开始训练模型。

## 4.4.6 使用模型

如何将训练好的模型用在新样本上呢？比如新建文件 deploy.prototxt，文件的内容和 train\_val.prototxt 类似，不同的是，deploy.prototxt 只定义了模型的结构，数据层的 data\_param 中不包括 source 数据路径。

下面使用 Caffe 的 Python 接口加载一个训练好的模型结构及权重：

```
caffe.set_mode_cpu() # or set_mode_gpu()

# 模型文件
model_def = '/.../deploy.prototxt'
# 训练结果
model_weights = '/.../gen/snapshot_train_iter_10000.caffemodel'
```

Blob 是 Caffe 中传递数据的封装对象，类似下面的代码预测一个 input 样本的标签：

```
net = caffe.Net(model_def, model_weights, caffe.TEST)

net.blobs['data'].data[...] = input # input 是待测样本
output = net.forward() # 前向迭代一次
output_prob = output['prob'][0] # 输出样本是各个类别的概率数值
```

更完整的 Caffe 训练实例参见第 7 章 Caffe 实例：狗狗品种辨别。



## 4.4.7 Caffe 的 Python 接口

Caffe 的 Python 接口，可以实现处理 I/O 数据、加载模型、前向/反向迭代、定义配置、绘制网络等功能。比如上面的配置文件，你也可以使用下面的代码生成。

生成模型文件：

```
from caffe import layers as L
from caffe import params as P

def logreg(lmdb, batch_size):
    n = caffe.NetSpec()
    # 载入数据，标签
    n.data, n.label = L.Data(batch_size=batch_size,
                             backend=P.Data.LMDB, source=lmdb,
                             transform_param=dict(scale=1. / 255),
                             ntop=2)

    # 定义逻辑分类模型运算
    n.ip1 = L.InnerProduct(n.data, num_output=10,
                           weight_filler=dict(type='xavier'))
    n.accuracy = L.Accuracy(n.ip1, n.label)
    n.loss = L.SoftmaxWithLoss(n.ip1, n.label)

    return n.to_proto()
```

生成 solver.prototxt 文件：

```
from caffe.proto import caffe_pb2

def solver(train_net_path, test_net_path):
    s = caffe_pb2.SolverParameter()

    # train test
    s.train_net = train_net_path
    s.test_net.append(test_net_path)

    s.test_interval = 1000 # 每 1000 个训练迭代测试一次
    s.test_iter.append(250) # 一次测试 250 个 batch

    s.max_iter = 10000

    # SGD (随机梯度下降) 的 learning rate (学习速率), SGD 在后面的章节会介绍
    s.base_lr = 0.01

    # SGD 学习速率的变化策略
    s.lr_policy = 'step'
    s.gamma = 0.1
    s.stepsize = 5000
```

```
# SGD 动量
s.momentum = 0.9
s.weight_decay = 5e-4

# 1000 迭代显示一次当前的 cross-validation 成绩
s.display = 1000

# snapshot 指对模型训练权重的保存操作
# 这里 10000 次迭代保存一下，训练过程中，命令行 ctrl+c 切断时也会自动保存一下
s.snapshot = 10000
s.snapshot_prefix = 'snapshot_train'
# CPU 模型
s.solver_mode = caffe_pb2.SolverParameter.CPU

return s
```

训练模型：

```
caffe.set_mode_cpu()
solver = caffe.get_solver(solver_path)
solver.solve()

accuracy = 0
batch_size = solver.test_nets[0].blobs['data'].num
test_iters = 250
for i in range(test_iters):
    solver.test_nets[0].forward()
    accuracy += solver.test_nets[0].blobs['accuracy'].data
accuracy /= test_iters

print("Accuracy: {:.3f}".format(accuracy))
```

## 4.4.8 回顾

和实验级的 Keras 相比，Caffe 有更详细、可定制模型参数选项以及更快的训练速度。Caffe 的使用整体可分为以下几步。

- (1) 准备数据 (hdf5/lmdb/leveldb)。
- (2) 准备模型配置文件，如 train\_val.prototxt。
- (3) 定义训练流程，如 solver.prototxt。

# 深度学习模型

传统的机器学习基于统计数学设计学习模型，通过分析特征数值规律完成任务目标。可以说，模型训练的基础是预处理好的样本特征向量。IRIS 花卉特征数据集如图 5-1 所示。

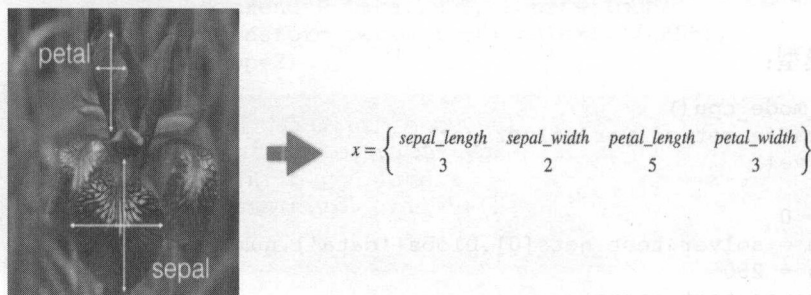


图 5-1 IRIS 花卉特征数据集

这种方式的最大优势在于模型的可解释性强，我们可以通过模型训练结果的权重直观地看到哪个事物特征发挥了多少程度的作用，而它的劣势同样在于过于依赖事物特征。

## 1. 并不是所有样本都有显式的可描述特征

比如一张图片、一段语音或者一些特殊事物。如何人工提取出这些样本的特征并不是一件容易的事情，如图 5-2 所示。

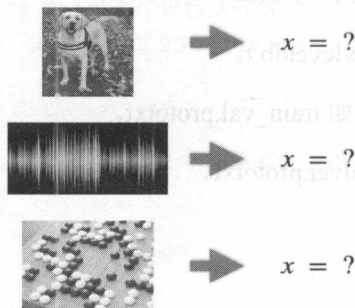


图 5-2 不容易提取特征的事物

## 2. 有些事物通过显式特征的方式不容易分辨

采集两个人 A、B 的录音信息，A 说“狗狗”和“痘痘”，B 说“宠物犬”。因为每个人都有自己的口音特征，所以对于传统的机器学习模型来说，A 的“狗狗”和“痘痘”表现得更相似，如图 5-3 所示；但在语义层，A 的“狗狗”和 B 的“宠物犬”才更相似。

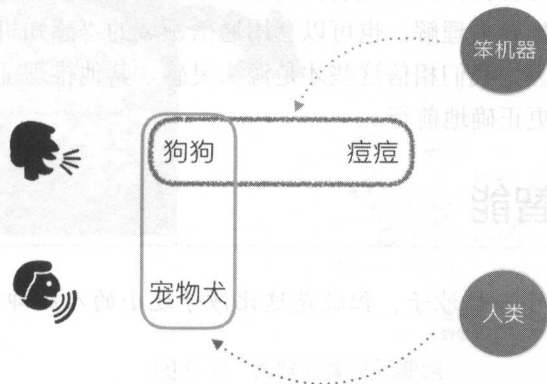


图 5-3 不易分辨的口音特征

这些任务对于人脑很容易完成，而通过传统的机器学习模型识别就变得很难。我们希望模型能够解决一些特征不好描述的任务问题。这一难题的解决来自另一个问题：一个 3 岁的孩子已经有足够的能力去理解自己看到的事物，但在孩子的早期教育中并没有人教他/她如何分析看到的事物的特征，那么他/她是如何学会这一技能的呢？孩子的视觉学习如图 5-4 所示。



图 5-4 孩子的视觉学习

(出自 TED, Fei-Fei Li: How we're teaching computers to understand pictures)

如果把人比作一台机器，那么自然界的数据可以说每时每刻都在训练我们的“脑模型”。按眼睛平均每 200ms 转动一次计算，一个 3 岁的孩子其实已经在真实的世界经验中学习过上亿张高质量的图片样本，于是他/她渐渐学会了识别事物。这一发现给了我们很大的启发：与其努力去解剖事物的特征，是不是可以通过喂给模型海量的样本，让模型自

动学会分析事物，自动挖掘出隐含的特征呢？

这就是深度学习模型灵感的源头：不去人工设计特征，而是通过模拟生物的学习方式，以“大数据+深度学习”的模式让模型自动剖析出样本的隐含特征。

在本章的小节中，我们将介绍深层神经网络模型。深度学习的核心原理有很多种解释方式，你可以用数学推导去理解，也可以套用通信领域的“感知机”去理解。本书更关注模型的生物学解释。因为我们相信这些才是源头灵感，其他推理证明都是在之后产生的，为了帮助科学家更好、更正确地前行。

## 5.1 解密生物智能

在宇宙中人就相当于一粒沙子，但就是这比沙子还小的人脑却在思考整个宇宙。

——亚里士多德

几乎每一个人都思考过这些问题：是什么控制了我的行为？为什么我的想法和他的想法总是不一样？所有类似的疑问最终都指向一个大悬疑：我们的大脑是如何工作的？

大脑的功能如图 5-5 所示。



图 5-5 多功能的大脑

世界充斥着无穷无尽的数据信息，文字、声音、景色，这些日常的数据统统输入大脑中，形成我们对世界的认知。大脑是如何处理这些信息的呢？

### 5.1.1 实验一：大脑的材料

首先，我们希望了解大脑是“什么材料”？



大脑的核心功能组织是大脑皮层，大脑皮层分为左右两个半球，在显微镜下观察，大脑皮层的基本建筑块料是神经元细胞，如图 5-6 所示。它包括一个特殊的细胞体和一个能把神经信息从一个神经元传递给下一个神经元的独特结构：突触。一个成年人类的大脑皮层的神经元数量大约在  $10^{10} \sim 10^{11}$  量级。

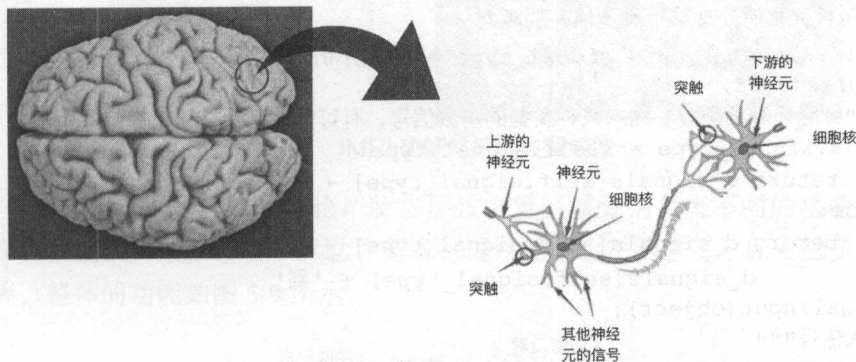


图 5-6 大脑的神经元细胞

对神经元进行刺激的实验发现，神经元有两种状态，静默状态和激活状态。当从其他神经元传递的输入信号满足特定条件时，神经元会进入激活状态，改变对外输出的生物电压，释放频率信号 (spike)，并将这一信号传递给下游的神经元。一个典型的神经元激活电压图如图 5-7 所示。

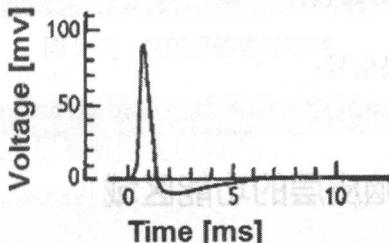


图 5-7 神经元激活电压 (出自 Wikipedia: Biological neuron model)

科学家同时发现，神经元可以通过调节突触，改变与其他神经元连接的突触的生物特性，相当于神经元携带着某种突触参数信息。

这个实验告诉我们：脑皮层是由神经元连接组成的，并且神经元携带着某些参数信息，具有“激活函数”的特性。

用代码模拟上述实验结果：

```
"""模拟由五感输入的刺激信号。对应行为：看 听 闻 触 嗅"""
d_signals = {
```

```

        '视觉信号': '看',
        '听觉信号': '听'
    }
}

class Neuron(object):
    """神经元"""
    def __init__(self, signal_type):
        # 神经元拥有处理某种类型信号的能力
        self.signal_type = signal_type # 突触携带的信息参数
    def spike(self, x):
        """神经元激活函数。输入某种类型的刺激信号，有可能激活神经元响应刺激"""
        if x.signal_type == self.signal_type:
            return d_signals[self.signal_type] + ':' + x.data
        else:
            return d_signals[self.signal_type] + ':' + '什么都没' + \
                d_signals[self.signal_type] + '到'

class SignalInput(object):
    """输入信号"""

    def __init__(self, signal_type, data):
        self.signal_type = signal_type
        self.data = data

```

当输入某些刺激信号时：

```

x = SignalInput('视觉信号', '一只猫在卖萌!')
print Neuron('视觉信号').spike(x)

```

输出神经元传递出的结果信号：

```
看：一只猫在卖萌！
```

## 5.1.2 实验二：探索脑皮层的功能区域

我们可以看到窗外的景色，听到美妙的音乐，甚至和朋友聊明星的八卦，幻想世界的末日。这些功能看起来完全不一样，脑皮层是如何完成这些不同的工作呢？

科学家们通过一些技术手段进一步研究了脑的功能，比如 fMRI（功能性核磁共振成像，如图 5-8 所示）对比实验。屏蔽实验者的视觉和其他刺激输入，对比让实验者抬起肢体和静躺状态下，脑皮层哪部分区域表现出明显的刺激作用；或者对比一些有特殊病理缺陷的病患大脑和正常的大脑在同一种相关刺激下脑活动的区别。

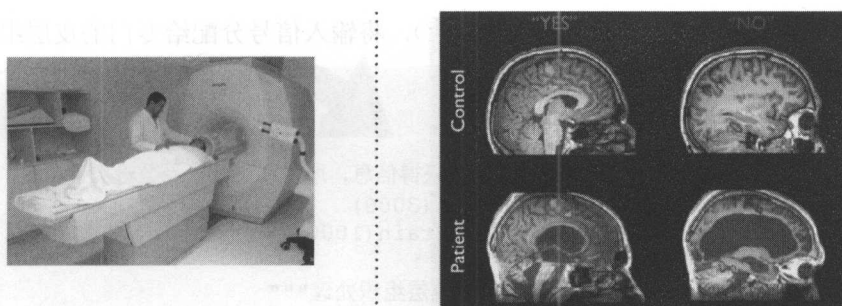


图 5-8 fMRI 功能性核磁共振成像

这些实验都指向一个明显的结论：脑皮层按位置区域分别负责不同的功能。实验结果表明：在大多数人类中，语言、意念、逻辑、理性主要由左脑掌管，而右脑负责形象思维和情感等。整体的功能如图 5-9 所示。

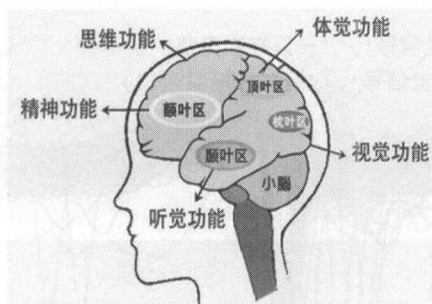


图 5-9 脑皮层的功能区域

这个实验给我们的启发：在脑皮层中，相同的神经元细胞按区域组成了不同的功能模块。用代码描述构造出视觉皮层和听觉皮层如下：

```
from collections import Counter
class VisualBrain:
    """视觉皮层"""
    def __init__(self, num):
        self.neurons = [Neuron('视觉信号') for i in range(num)]
    def process(self, x):
        """处理信号"""
        print '视觉皮层在处理：' + x.data
class AuditoryBrain:
    """听觉皮层"""
    def __init__(self, num):
        self.neurons = [Neuron('听觉信号') for i in range(num)]
    def process(self, x):
        """处理信号"""
        print '听觉皮层在处理：' + x.data
```

大脑根据传递过来的渠道（眼睛 or 耳朵），将输入信号分配给专门的皮层组织处理：

```
class Brain:
    """脑皮层"""
    def __init__(self):
        # 分配神经元数量，人脑更依赖视觉输入获得信息，所以视觉神经元数量应该更多
        self.visual_model = VisualBrain(3000)
        self.auditory_model = AuditoryBrain(1000)
    def process(self, x):
        """根据不同的传入信号，传递给不同的皮层组织处理"""
        result = {
            '视觉信号': lambda x: self.visual_model.process(x),
            '听觉信号': lambda x: self.auditory_model.process(x),
        }[x.signal_type](x)
```

比如同时输入两种信号：

```
brain = Brain()
x_see = SignalInput('视觉信号', '一只猫在卖萌!')
x_hear = SignalInput('听觉信号', '猫咪在喵喵叫!')
brain.process(x_see)
brain.process(x_hear)
```

输出：

```
视觉皮层在处理：一只猫在卖萌！
听觉皮层在处理：猫咪在喵喵叫！
```

### 5.1.3 实验三：不同的皮层组织——区别在于函数算法

是什么导致了脑皮层这一功能区域的划分？

科学家找来志愿者，通过小的电极连接到显示仪器，采样了不同皮层区域的神经元的信号数据。有趣的事情发生了，不同功能的神经元细胞对于同样的输入信号产生出不同的响应函数，如图 5-10 所示。

图中每个方框内是采集的一组特定区域的神经元的响应函数（这些都是从实验室流出的真实图片数据）。通过观察我们可以发现：

（1）除去一些噪声干扰，相同功能的神经元细胞对刺激表现出的响应函数几乎完全相同，如图 5-11 所示。



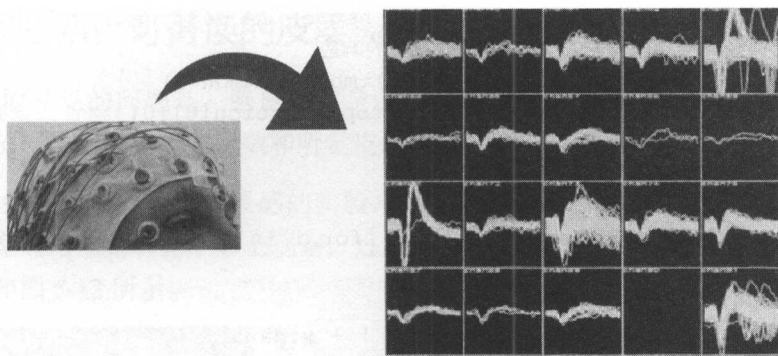


图 5-10 不同皮层区域的神经元刺激——响应函数图

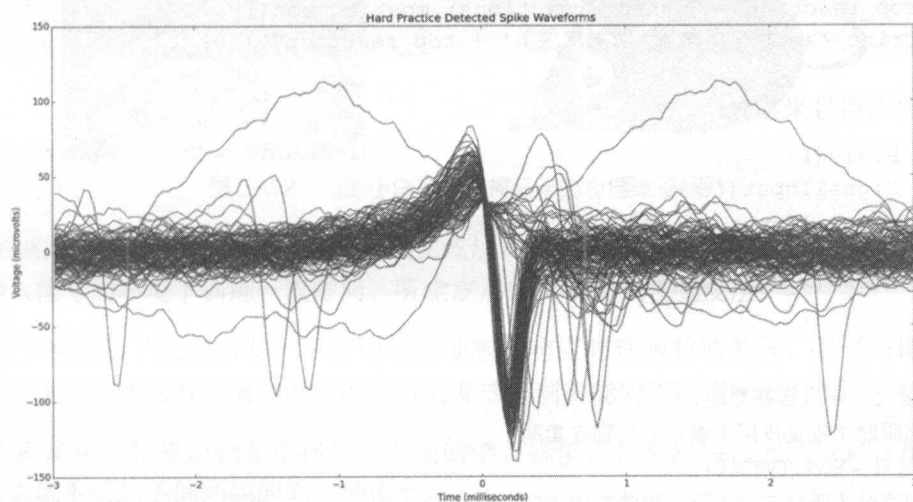


图 5-11 同一皮层区域的神经元刺激——响应函数图

(2) 不同皮层区域的神经元有着不同的响应函数。

实验结论：对输入的生物刺激，大脑不同皮层模块的神经元细胞有着不同的响应处理函数。

如下代码所示，每个皮层组织由一群相同响应的神经元投票决定输出信号处理结果：

```
class VisualBrain:
    """视觉皮层"""
    def __init__(self, num):
        self.neurons = [Neuron('视觉信号') for i in range(num)]
    def process(self, x):
        """处理信号"""
        print '输入' + x.signal_type + ': ' + x.data
```



```

        reactions = [neuron.spike(x) for neuron in self.neurons]
        # 一群神经元集体响应投票，打印票数最高的响应
        top_reaction = Counter(reactions).most_common(1)
        print '>> 我在用眼睛（视觉皮层）' + top_reaction[0][0]
class AuditoryBrain:
    """听觉皮层"""
    def __init__(self, num):
        self.neurons = [Neuron('听觉信号') for i in range(num)]
    def process(self, x):
        """处理信号"""
        print '输入' + x.signal_type + ': ' + x.data
        reactions = [neuron.spike(x) for neuron in self.neurons]
        # 一群神经元集体响应投票，打印票数最高的响应
        top_reaction = Counter(reactions).most_common(1)
        print '>> 我在用耳朵（听觉皮层）' + top_reaction[0][0]

```

当输入信号来临时：

```

brain = Brain()
x_see = SignalInput('视觉信号', '一只猫在卖萌!')
x_hear = SignalInput('听觉信号', '猫咪在喵喵叫!')
brain.process(x_see)
brain.process(x_hear)

```

输出：

```

输入视觉信号：一只猫在卖萌！
>> 我在用眼睛（视觉皮层）看：一只猫在卖萌！
输入听觉信号：猫咪在喵喵叫！
>> 我在用耳朵（听觉皮层）听：猫咪在喵喵叫！

```

如果我们将信号输入到对应错误的皮层组织时：

```

brain.auditory_model.process(x_see)
brain.visual_model.process(x_hear)

```

输出：

```

输入视觉信号：一只猫在卖萌！
>> 我在用耳朵（听觉皮层）听：什么都没听到
输入听觉信号：猫咪在喵喵叫！
>> 我在用眼睛（视觉皮层）看：什么都没看到

```

### 5.1.4 实验四：可替换的皮层模块——神经元组成的学习模型

一个显而易见的问题是，脑皮层都是由神经元细胞组成的，那么同样是神经元细胞，为什么视觉皮层的神经元响应函数和听觉皮层的神经元的响应函数会不同呢？

谜底的揭晓源自一个偶然灵感的实验：科学家把小白鼠的听觉中心的神经和耳朵通路剪断，视觉通路接到听觉中心上观察。过了几个月发现，小白鼠可以通过听觉中心处理视觉信号，如图 5-12 所示。

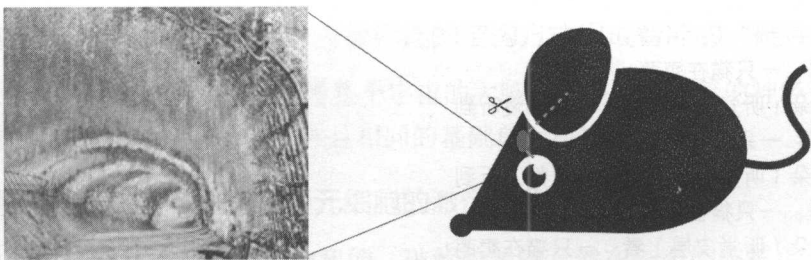


图 5-12 训练小白鼠听觉神经元的视觉处理能力

也就是说，小白鼠的听觉皮层和视觉皮层其实没有任何生理结构上的区别，只要在视觉输入信号的环境下训练一段时间，听觉皮层就可以变成视觉皮层！

这个实验给了我们启发：不同神经元细胞的响应处理函数的差异，是通过不同的数据训练（生物刺激信号）演变而成的，进而变成不同功能的皮层组织。

也就是说：脑皮层就是由神经元组成的学习模型，各个皮层功能区域只是经过不同的数据训练形成的不同功能的同一种模型。

用代码模拟神经元的可变性，假设每个神经元有 1% 的概率领悟如何处理当前输入的信号（实际肯定远低于这个几率数量级）。

```
import random
class Neuron(object):
    """可被训练的神经元"""
    def __init__(self, signal_type):
        # 神经元拥有处理某种类型信号的能力
        self.signal_type = signal_type # 突触携带的信息参数，且可被训练改变
    def spike(self, x):
        """输入某种类型的刺激信号，有可能激活神经元响应刺激"""
        if random.random() < 0.01: # 假设神经元有 1% 的几率被训练改变
            self.signal_type = x.signal_type
        if x.signal_type == self.signal_type:
            return d_signals[self.signal_type] + ':' + x.data
```

```

else:
    return d_signals[self.signal_type] + ':' + '什么都没' + \
        d_signals[self.signal_type] + '到'

```

连着输入多次，会发现神经元的处理类型发生了变化：

```

brain = Brain()
for i in range(100):
    brain.auditory_model.process(x_see)

```

输出：

```

.....
输入视觉信号：一只猫在卖萌！
>> 我在用耳朵（听觉皮层）听：什么都没听到
输入视觉信号：一只猫在卖萌！
>> 我在用耳朵（听觉皮层）听：什么都没听到
输入视觉信号：一只猫在卖萌！
>> 我在用耳朵（听觉皮层）看：一只猫在卖萌！
输入视觉信号：一只猫在卖萌！
>> 我在用耳朵（听觉皮层）看：一只猫在卖萌！
.....

```

可以看到，多次迭代训练后，听觉皮层领悟到如何“看”。

### 5.1.5 模拟神经元

更进一步，科学家发现神经元是可以训练的。神经元通过改变与其他神经元连接的突触的生物特性，从而达到给不同连接分配“权重”的效果——相当于模型中的参数，如图 5-13 所示。

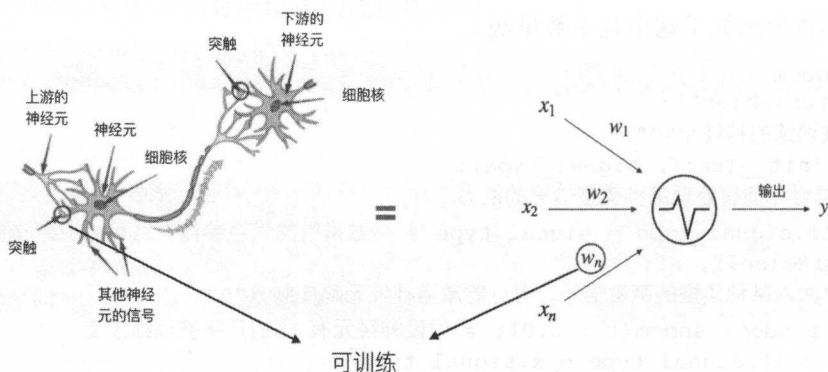


图 5-13 工程模拟神经元的输入输出

图中  $x$ 、 $y$  对应的生物学意义就是神经元的生物刺激的频率信号 (spike);  $w$  的生物学意义就是突触携带的“生物权重”。当  $w$  为正数发挥作用时, 相当于突触产生正电流增强神经元信号;  $w$  为负数发挥作用时, 就是突触产生负电流抑制神经元信号。这就是工程上设计的对神经元的模拟结构: 人工神经元。

### 5.1.6 生物结构带来的启发

上面一系列的实验带给我们的核心启发有两条。

#### 1. 生物以一个简单的元细胞, 重复相连的结构方式组成智能的“硬件”

只有 1.4 立方, 却让我们能够思考整个宇宙的大脑, 只是一堆重复的神经元相连而已。一个复杂的模型结构, 是由一群简单并且相同的基础单元结构组成。

#### 2. 智能的关键, 藏在一个神经元细胞的激活函数中

对于像人脑这样复杂高端的生物智能, 破解的关键竟然在一个细胞核中, 这听起来不可思议! 其实, 生物的奥秘, 从来都隐藏在其组成的基本单元中。比如多年以前你问一个生物学家人体的组织结构, 他会说这个问题很复杂, 蛋白质是如何如何合成的, 从哪里获得构造肽键的能量; 而今天的分子生物学家对此则会说, “关键不在那, 在于组装氨基酸序列的指令编码; 让能量见鬼去吧, 它自己会想办法的。” 神经元—DNA 类比图如图 5-14 所示。

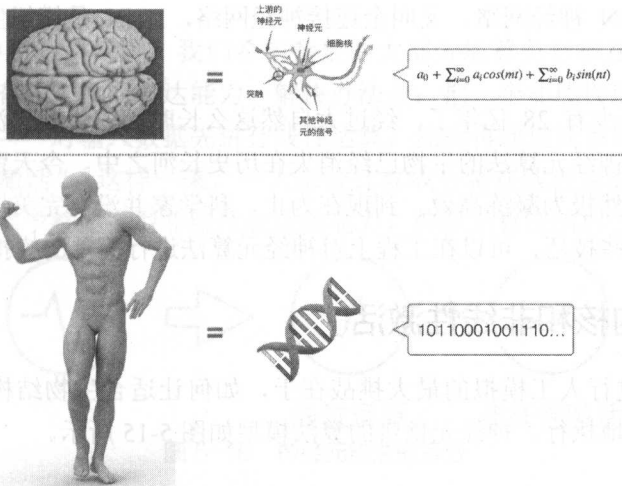


图 5-14 神经元—DNA 类比图

备注: 图 5-14 中我们仅仅是用一个无穷的表达式代表神经元激活算法的表达式, 和破解遗传编码一样, 对神经元算法的破解至今还在进行中。

从信息量的角度来看，庞大如我们的身体组织，并不比一条 DNA 载体上包含的信息量更多。关于生物体的完整描述都已经按生物自己的编程语言写在了遗传基因里。同样的，关于生物智能的关键描述也按生物的语言方式藏在神经元细胞的激活算法中，而所有的智能表现都是输入+核心算法重复执行的结果。

借鉴生物灵感，人工设计神经元算法，以简单重复的方式构造模型结构，这就是人工神经网络模型。

### 5.1.7 回顾

本节介绍了人工神经网络模型的生物学解释，更多的细节将在后续介绍。

- 脑皮层由神经元组成。
- 神经元携带的激活“算法”就是智能的关键。
- 设计神经元算法，构造神经网络——人工神经网络模型。

## 5.2 DNN 神经网络模型

创新就是把各种事物整合到一起。

——史蒂夫·乔布斯《追随你的心》

本节介绍 DNN 神经网络，又叫全连接神经网络，DNN 是模拟生物神经元网络的网络模型。

生物的历史至少有 28 亿年了，经过大自然这么长时间的优胜劣汰，那些携带着低效的、不适应自然的神经元算法的生物已经消失在历史长河之中。今天留下的生物，神经元携带的激活算法必然极为凝练高效。到现在为止，科学家并没有完美地破解这一算法，但我们已经掌握了一些技巧，可以在工程上对神经元算法进行简单的模拟。

### 5.2.1 线性内核和非线性激活

对生物结构进行人工模拟的最大挑战在于，如何让适合生物结构执行的算法下放在计算机结构中高效地执行。神经元携带的算法模型如图 5-15 所示。



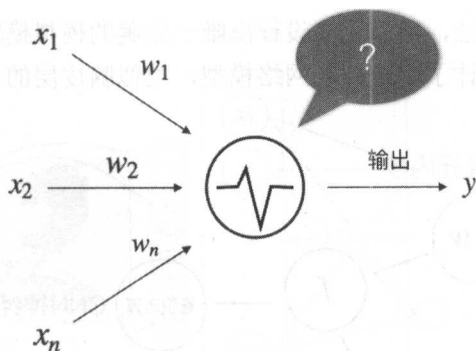


图 5-15 神经元携带的算法模型

首先我们知道：神经元的算法一定是非线性的。因为如果神经元的响应函数是线性算法，由于线性之间的运算都可以变成大矩阵的乘法，那么无论大脑有多少神经元在通信交互，其整个思考过程会变得和一系列乘法计算完全等价，我们将不可能拥有跳跃式思考的能力。

然而，计算机偏爱线性计算。线性运算本质上就是矩阵运算，计算机有专门的硬件处理线性运算：GPU。在数值计算中，科学家也更欢迎线性运算，其原因如下：

- 线性运算非常高效，因为它就是大矩阵的乘法运算。
- 线性运算稳定，某个输入分量的波动对结果的影响不会太大。
- 导数形式非常理想，在梯度下降时这一点非常重要。

所以，在对神经元建模时，我们希望参数的大部分运算负担放在线性函数中执行，但模型整体上又拥有非线性的表达能力。解决办法是：把一个非线性运算拆成线性内核与非线性激活的叠加——对输入数据先进行线性运算，然后再通过简单的非线性函数对线性运算的结果进行“激活”。这样一来，一大部分计算成分将在线性函数中高效完成，接着又通过非线性函数赋予整体模型非线性表达的能力。神经元算法的拆分如图 5-16 所示。

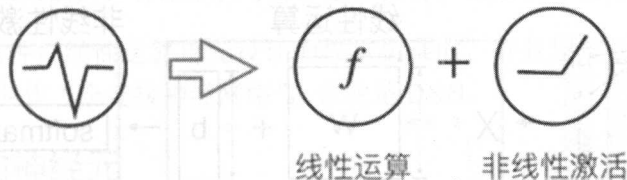


图 5-16 神经元算法的拆分

### 5.2.2 DNN、CNN、RNN

通过“线性内核+非线性激活”设计神经元是一个非常重要的思想。由于我们并没有

完美破解神经元的激活算法，所以无法设计出唯一完美的模拟模型。在深度学习中，我们根据不同的功能需要，设计了多种神经网络模型，类似脑皮层的不同组织。可更换的神经元内核如图 5-17 所示。

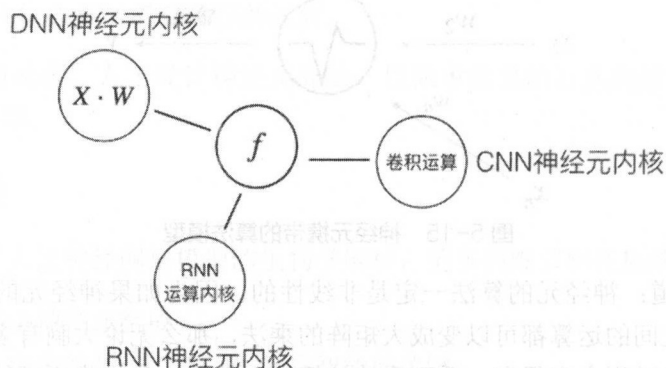


图 5-17 可更换的神经元内核

我们把其中的线性运算设计成权重相乘，这就是本章后续将登场的 DNN 模型的神经元；也可以换成卷积运算（卷积是一种线性运算，第 6 章介绍），这就是 CNN 模型的神经元。甚至设计一些运算单元模拟“记忆”的特性，这就是 RNN 的神经元（第 8 章介绍）。

DNN（Deep Neuron Network 深度神经网络）神经元就是权重矩阵的乘法运算和激活运算的叠加。接下来继续以 MNIST 为例，通过代码实现展开介绍这一神经网络模型。

### 5.2.3 逻辑分类：一层神经网络

逻辑分类可以被视为一层 DNN 神经元组建的全连接神经网络（1 层 DNN），背后的原因是逻辑分类满足以  $X \cdot W + b$  为线性的运算内核，以 Softmax 函数（二分类中使用 Sigmoid 函数）为非线性激活函数的构造方式。逻辑分类的代码实现参见 4.3 深度学习框架快速上手-Keras 完成逻辑分类。MNSIT 逻辑分类如图 5-18 所示。

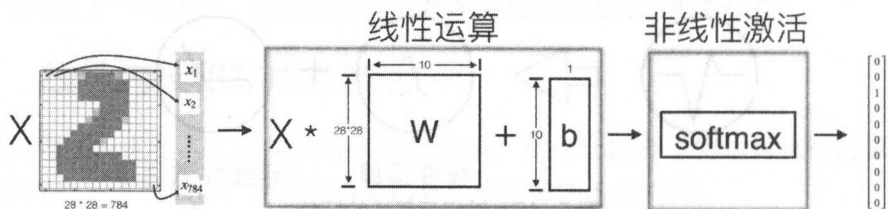


图 5-18 MNSIT 逻辑分类

简化图形之后，将逻辑分类以神经元模型的链接表示出来，输入图片数据的神经层叫作“Input Layer（输入层）”，输出类别的神经层叫作“Output Layer（输出层）”，如图 5-

19所示，逻辑分类的输出是和每个输入分量相连的。

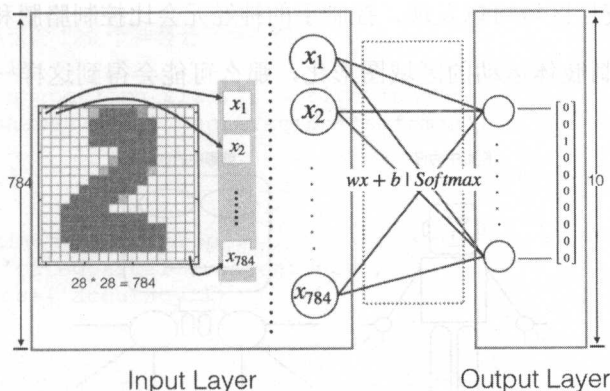


图 5-19 MNIST 逻辑分类连接图

在逻辑分类中，输入数值是归一化的图片像素值，这里输入样本的“特征”就是像素对应的位置。如果用二分类模型观察一下模型学到对应每个数字的权值矩阵，可以发现一些很有意思的事情。比如，对是否是数字“0”的图片进识别时，训练好的模型的权值  $W$  类似图 5-20 所示。

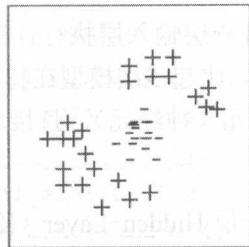


图 5-20 数字 0 的权重图

说明：“+”表示正数权值，当有像素在对应像素位置时，模型认为图片更可能为 0；反之，“-”表示负数权值，当有像素在对应像素位置时，模型认为图片更不可能为 0。

像逻辑分类这种，线性运算单元设计为权重相乘的，并且层与层之间的神经元全部相连的神经网络叫作“全连接神经网络”，也就是 DNN。

## 5.2.4 更多的神经元

在对神经元进行的生物研究中，科学家还发现了一些有意思的现象。在对脑皮层控制中枢研究时，科学家发现皮层区域的神经元越多，对应的肢体控制能力越丰富。比如说，从外形体积上看，“手”比腿、胳膊小得多，但我们对手的控制能力却丰富得多。我们可

以用手编程、织毛衣、操作精密的仪器，相对地，对胳膊和腿的控制只限制在简单的弯曲、移动等。所以在脑皮层区域可以发现，控制手的神经元会比控制胳膊和腿的多得多。

如果将脑部控制肢体运动的区域图形化，那么可能会得到这样一个奇怪的图，如图 5-21 所示。

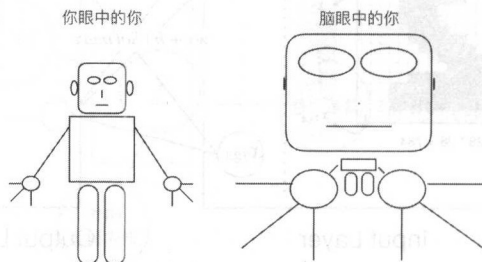


图 5-21 脑眼中的你

对于生物而言，神经元数量越多，处理能力越强大。对于工程模型同样如此，神经元越多，模型能力越强大，那么训练时越容易发生过拟合。

### 5.2.5 增加 Hidden Layer（隐层）

之前的例子中，逻辑分类只有一层输入层执行计算，我们可以增加一个中间层容纳更多的神经元，来增强模型的能力。比起浅层模型在特征工程和模型工程的各种尝试，神经网络可以通过增加更多的计算单元（神经元），直接增强模型的能力。这正是由单一算法组成的复杂模型的美妙之处。

如图 5-22 所示，我们增加了一层 Hidden Layer（隐层），包含 128 个神经元节点。这里的“128”是一个可调的参数，如同在浅层模型中 GridSearch 暴力搜索优化的模型参数一样。一般会将数量设置的略高于任务的需要，然后通过调试解决过拟合问题。

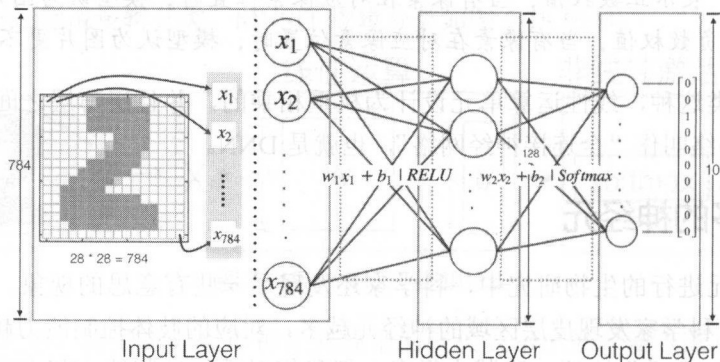


图 5-22 含隐层的 DNN

输入:

```
# 创建模型, 增加一层包含 128 个神经元节点的隐层
model = Sequential([
    Dense(128, input_shape=(img_size,), activation='relu'),
    Dense(10, input_shape=(128,), activation='softmax'),
])

# 编译模型
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# 训练
model.fit(X_train, Y_train,
        batch_size=batch_size, nb_epoch=nb_epoch,
        verbose=1, validation_data=(X_test, Y_test))

# 测试
score = model.evaluate(X_test, Y_test, verbose=0)
print('accuracy: {}'.format(score[1]))
```

输出:

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [=====] - 3s - loss: 0.3297 - acc:
0.9098 - val_loss: 0.1880 - val_acc: 0.9442
Epoch 2/10
60000/60000 [=====] - 3s - loss: 0.1549 - acc:
0.9553 - val_loss: 0.1377 - val_acc: 0.9592
Epoch 3/10
60000/60000 [=====] - 2s - loss: 0.1123 - acc:
0.9678 - val_loss: 0.1137 - val_acc: 0.9654
Epoch 4/10
60000/60000 [=====] - 3s - loss: 0.0878 - acc:
0.9742 - val_loss: 0.0966 - val_acc: 0.9710
Epoch 5/10
60000/60000 [=====] - 2s - loss: 0.0721 - acc:
0.9788 - val_loss: 0.0881 - val_acc: 0.9738
Epoch 6/10
60000/60000 [=====] - 2s - loss: 0.0612 - acc:
0.9819 - val_loss: 0.0838 - val_acc: 0.9745
Epoch 7/10
60000/60000 [=====] - 2s - loss: 0.0521 - acc:
0.9851 - val_loss: 0.0814 - val_acc: 0.9764
Epoch 8/10
60000/60000 [=====] - 3s - loss: 0.0447 - acc:
0.9871 - val_loss: 0.0804 - val_acc: 0.9757
Epoch 9/10
```



```
60000/60000 [=====] - 3s - loss: 0.0387 - acc:
0.9891 - val_loss: 0.0796 - val_acc: 0.9769
Epoch 10/10
60000/60000 [=====] - 3s - loss: 0.0342 - acc:
0.9899 - val_loss: 0.0800 - val_acc: 0.9768
accuracy: 0.9768
```

## 5.2.6 ReLu 激活函数

注意，在模型的隐层中，激活函数使用 `activation='relu'`。这是一个什么样的函数呢？

ReLU 函数代码：

```
def relu(x):
    return x * (x > 0)
```

ReLU 函数如图 5-23 所示。

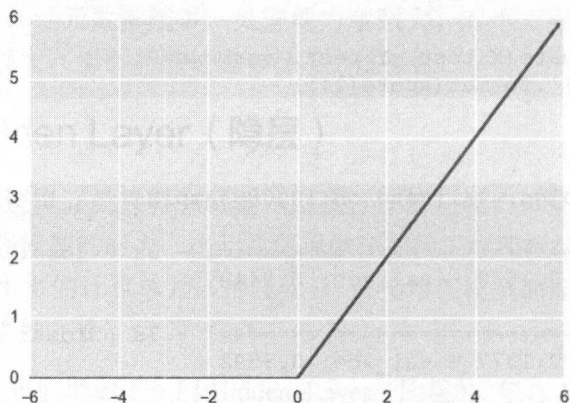


图 5-23 ReLu 函数

ReLU 左边永远是 0，右边相当于  $y = x$ 。那么为什么要用 ReLu 函数呢？

主要原因：快。采用 Softmax/Sigmoid 等函数，算激活函数时涉及指数运算，在求梯度导数时则涉及除法运算，神经元很多时计算量还是比较庞大的。而 ReLu 函数的计算很简单，而且导数是常量，机器算起来非常快。也有些论文指出，ReLU 函数更接近生物的神经激活性质，以及工程上，ReLU 计算时的一些数学特性对解决神经网络的过拟合有好处。

汇总一下，对于一张输入图片样本，全连接神经网络完整的计算流程如图 5-24 所示。

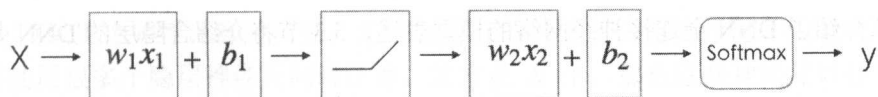


图 5-24 DNN 计算流程

有很多其他激活函数，有兴趣的读者可以参考维基百科：[https://en.wikipedia.org/wiki/Activation\\_function](https://en.wikipedia.org/wiki/Activation_function)。

如果你选择使用 ReLu 函数激活，那么调试时需要注意小心设置学习速率，不要让过多的神经元处于 ReLu 函数的左边死亡状态。

### 5.2.7 理解隐层

如何理解隐层的学习过程呢？在输入层的神经网络中（逻辑分类），我们可以看到，学习到的权值矩阵从某种程度上刻画了图片的特征。但隐层又做了什么呢？

逻辑分类使用的输入样本的“像素位置”特征其实是一种很底层的描述特征，DNN 模型的隐层在此基础上，自动寻找了合理的解读特征。如图 5-25 所示。我们并不知道隐层挖掘的特征具体含义是什么（这也是为什么称之为“隐层”的原因），只是知道隐层在模型训练的过程中重新构建了某些特征。在梯度下降的过程中，模型参数不断逼近最优的参数组合的同时，隐层不断地重构其表达的特征数据，这些特征朝着最能够表示上一层数据集的特征方向上不断逼近。打个比方，对于图片“2”的识别，隐层中可能在这么做。

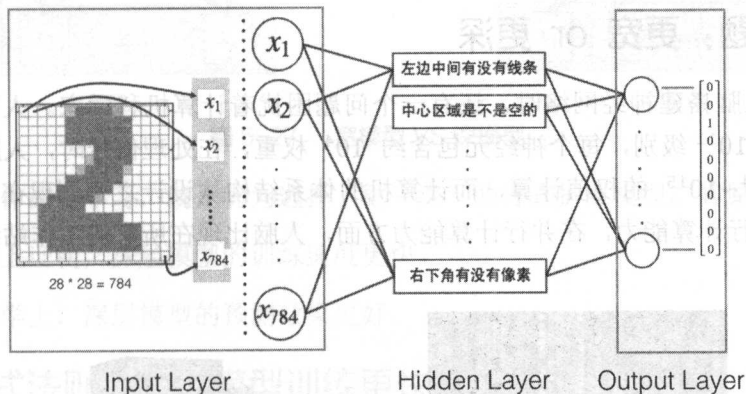


图 5-25 DNN 隐层特征

还有一点值得说明，神经网络本质上是弱人工智能的实现，计算任务背后的意义还是要由我们引导。当训练标签的  $y$  设置成图片标签数字时，每一层神经网络都会努力寻找标签数字的特征。更换  $y$  的意义，比如设置成“图片中的背景图是什么颜色”时，那么神经网络的就会朝另一个方向寻找特征。

本节介绍了 DNN 全连接神经网络的模型表述，5.3 节将介绍含隐层的 DNN 是如何训练的。

## 5.2.8 回顾

- 人工神经元的算法：线性内核+非线性激活。
- DNN（全连接神经网络）使用权值乘法作为线性运算，ReLU 函数激活。
- DNN 包含三层：输入层、隐层、输出层。
- 使用 Caffe 和 Keras 训练 DNN。
- 理解 DNN：隐层在模型的训练过程中自动发掘的解读样本的特征。

## 5.3 神经元的深层网络结构

我们每个人是由 10 的 28 次方个原子组成，而我们的宇宙共有 10 的 78 次方个原子。我们的世界虽然庞杂，却是由六个数精确谐调而成。

——马丁·里斯《六个数：塑造宇宙的深层力》

本节将扩展神经网络，让模型更强。当扩展神经网络的时候，我们选择面向深层的设计。

### 5.3.1 问题：更宽 or 更深

在模拟人脑搭建神经网络时，还有一个问题困扰着计算机科学家。人脑的神经元数量约为  $10^{10} \sim 10^{11}$  级别，每个神经元包含约  $10^4$  权重，在处理信号时，人脑可以在 1ms 内进行约  $10^{14} \sim 10^{15}$  的权值计算，而计算机的体系结构从设计之初到现在为止，赋予的都是强大的串行计算能力，在并行计算能力方面，人脑比现在所有的工作站都强。如图 5-26 所示。

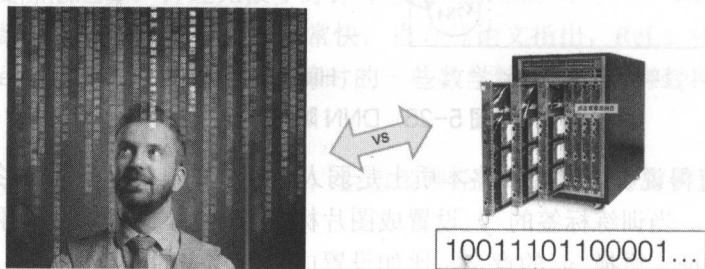


图 5-26 人脑 VS 计算机

一个模拟生物并行计算的模型，怎样在串行的机器上快速执行呢？假设有足够的数据，想要使用很多个隐层神经元同时计算，比方说  $N$  个，那么这些神经元以什么样的扩展方式接入模型呢？

摆在我们面前的有两条路：（1）在一个隐层中加入更多的神经元，让模型变得更“宽”；（2）加入更多的隐层，让模型变得更“深”。如图 5-27 所示。

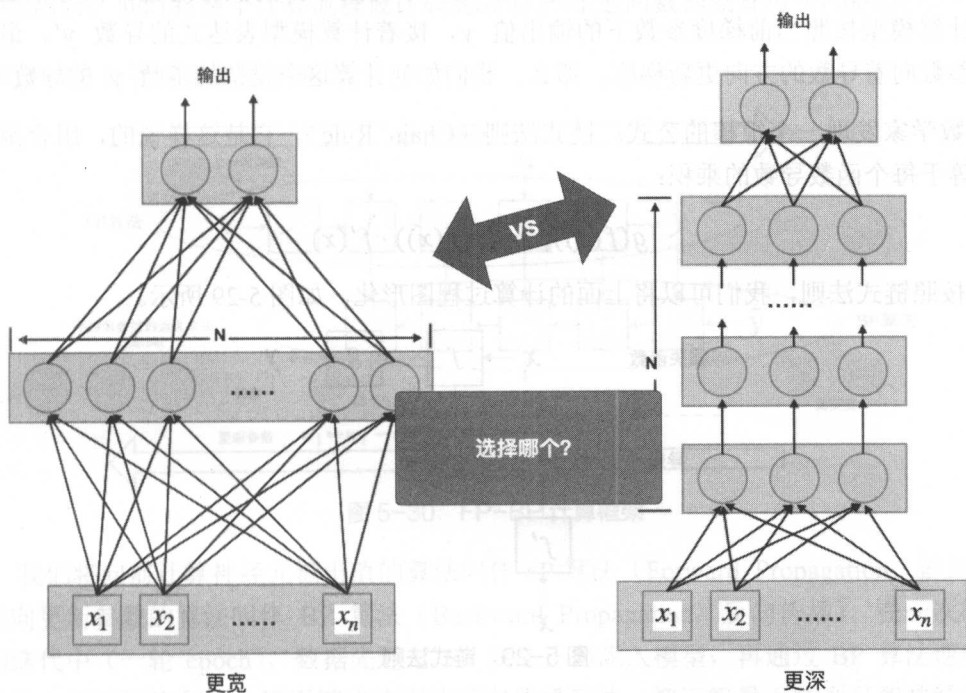


图 5-27 宽模型 VS 深模型

之所以叫作深度学习，就是因为选择了更“深”的方式扩展模型。其原因有两个：

（1）数学上证明，深层模型的训练速度更快。

（2）生物学上，深层模型的预测效果更好。

### 5.3.2 链式法则：深层模型训练更快

如果选择更深的模型结构，输入数据集  $X$ ，那么整个框架的计算过程将按函数输入输出组合起来，用数学表达式可以表示为  $y = \text{softmax}(\dots \text{fadd}(\text{fmulti}(\text{frelu}(\text{fadd}(\text{fmulti}(x))))))$ 。

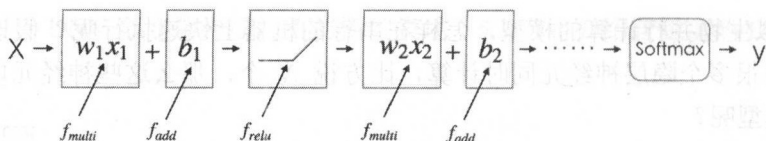


图 5-28 深层模型的数学表达式

接下来用梯度下降优化模型的参数。回忆一下梯度下降的执行过程：在每个迭代中，首先计算模型按照当前梯度参数下的输出值  $y$ ，接着计算模型表达式的导数  $y'$ ，最后让模型参数向着导数的方向更新梯度。那么，我们如何计算这个复杂的函数  $y$  的导数呢？

数学家发现一个很棒的公式：链式法则（Chain Rule）。它是这样说的，组合函数的导数等于每个函数导数的乘积：

$$[g(f(x))]' = g'(f(x)) \cdot f'(x)$$

按照链式法则，我们可以将上面的计算过程图形化，如图 5-29 所示。

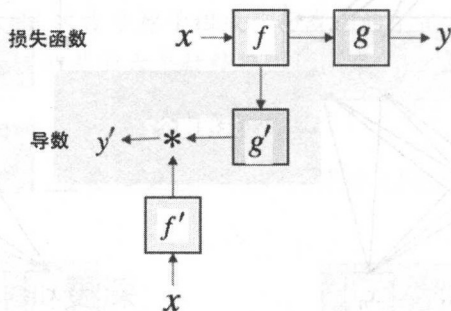


图 5-29 链式法则

在逻辑分类中（单层神经网络），我们以交叉熵作为损失函数，计算公式为  $loss(w, b) = -\frac{1}{M} \sum_i D(y_i, p_i)$ ，对权重参数  $w_j$  损失函数的偏导数形式非常简单： $\frac{\partial}{\partial w_j} loss(w_1, w_2) = \frac{1}{M} \sum (p_i - y_i) x_i$ ，其中  $i$  是样本标号， $p_i = softmax(score(wx_i + b))$ 。也就是说，在计算损失函数和导函数的过程中都需要重用  $p_i$  的计算值：

#  $X$  是训练样本矩阵， $w$  是权重向量， $b$  是偏置向量， $y$  是真实标签矩阵

```
def loss_func(X, w, b, y):
    """损失函数"""
    s = score(X, w, b)
    y_p = softmax(s)
    return -np.mean(cross_entropy(y, y_p))
```

```
def d_loss_func(X, w, b, y, w_i):
    # 损失函数对 w 的偏导数
    s = score(X, w, b)
```



```
y_p = softmax(s)
return np.mean(w_i * (y_p - y))
```

同样地，多层神经网络的组合函数中，每一层的预测值都会在计算偏导数时复用，同时模型的中间函数都是诸如  $f_{add}$ 、 $f_{multi}$ 、 $f_{relu}$  等，它们的计算非常简单。根据链式法则提供的复合运算的导数计算方式，我们可以将一个复杂的梯度计算过程变成一系列简单计算的组合，同时将整个计算流程设计成高效的正向/逆向数据流管道（Pipeline）。

FP-BP 计算框架如图 5-30 所示。

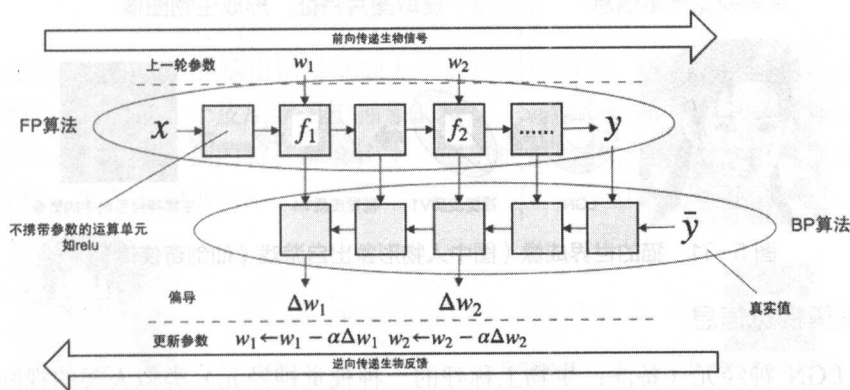


图 5-30 FP-BP 计算框架

我们将向前计算神经元输出值的算法叫作 FP 算法（Forward Propagation，前向传播），将逆向更新参数的算法叫作 BP 算法（Backward Propagation，反向传播）。在一次梯度下降的迭代中（一轮 epoch），数据先通过 FP 算法正向流入模型，再通过 BP 算法逆向计算偏导数，更新梯度参数。运用链式法则的反向传播算法，把运算量下降到只和神经元数目本身成正比  $O(n)$ 。

还有一点值得说明，从生物的角度看，这套执行流程可以理解为：在模型训练过程中，FP 算法一层一层向前传递生物“信号”，而 BP 算法逆向一层一层向后计算，通过更新梯度的方式提供生物“反馈”。

总的来说，在链式法则支持下，选择深层模型可以做成 I/O 管道结构，而宽层的模型的参数计算过程没有好的拆解方式。在计算结构方面，深层模型比宽层模型更高效。

### 5.3.3 生物：深层模型匹配生物的层级识别模式

使用深层模型的另一个重要原因就是它使模型的成绩更加理想，这一点在工程中已经反复证实，这里给出这一结果在生物学上的解释。

## 1. 生物实验：猫咪如何看世界

人脑有海量的神经元，它们是如何组织在一起完成工作的呢？科学家通过研究猫是如何看世界的，进而分析人脑的视觉形成。

研究发现，猫分为两步看到外界事物：（1）采集视觉信息，（2）提取视觉特征形成图像。下面将通过一个模拟的例子展开。

猫的世界成像如图 5-31 所示。

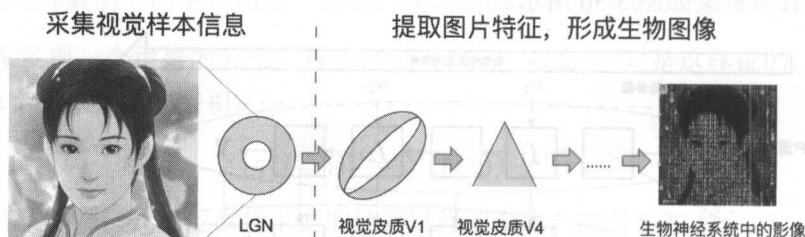


图 5-31 猫的世界成像（图中人物形象出自游戏《仙剑奇侠传》）

## 2. 采集视觉信息

猫的 LGN 神经元（备注：生物上称呼的一种视觉神经元）类似人类的视网膜神经元，只起了传递信号的作用，非常类似工程上将图片像素解析成 RGB 数值矩阵，LGN 神经元将图片解析成一组对应的生物刺激信号，如图 5-32 所示。

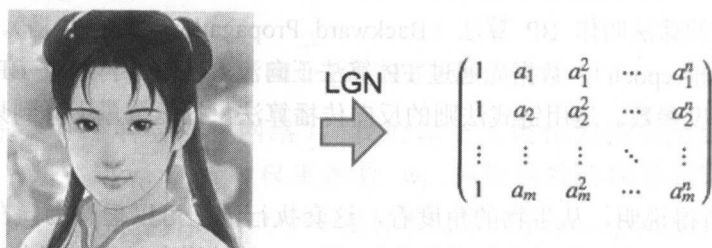
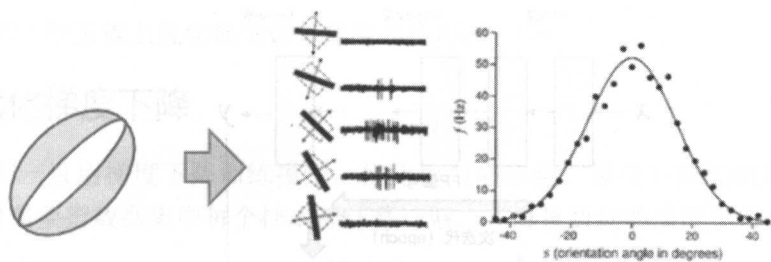


图 5-32 猫的 LGN 神经元原理

## 3. 提取视觉特征，组合成像

视觉信息以生物刺激的方式输入到神经皮质中。科学家对视觉皮质 V1 进行研究发现，V1 神经元只能识别非常粗浅的特征，比如，一个图片的某种边缘、线条等。

如图 5-33 所示，当每个神经元处理一小部分区域（视野），且视野内的图像在指定的方向上（黑色短线）有强视觉刺激时，该 V1 神经元输出强烈的生物电流信号（高频，竖线密集），视觉刺激越偏离指定方向，输出信号越随之减弱。



V1

Gaussian tuning curve of a cortical (V1) neuron

图 5-33 V1 神经元

视觉皮质 V4 在 V1 的输出信号基础上学习识别出更广、更抽象的特征，比如图片区域局部的特征；然后下一个皮层学习更抽象的特征，逐层向后……每下一层神经元在上一层输出信号的基础上，学到的特征都会比上一层更抽象、更全局，最终组合形成生物神经系统中的图像，如图 5-34 所示。

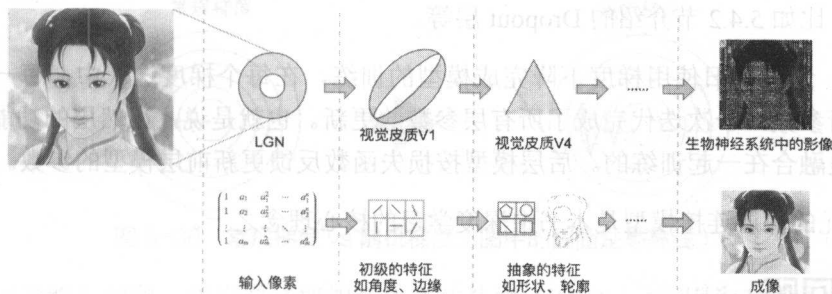


图 5-34 成像原理

整体上看，生物神经系统以层级特征的方式，将外界物理光学信号组成的图像编码成生物神经系统中的电流信号组成的视觉影像。生物智能以层级特征的处理模式在对人脑的研究中得到证实，在核磁共振成像实验中发现脑皮层也有类似的层级结构。

实验的结论是：生物神经系统以面向“层”的方式识别事物，每一层在上一层识别的基础上提取更抽象的特征。也就是说，生物本身也是深层的结构模式，而非宽层。

这一点无疑给了我们提示，在面向“更宽 or 更深”的问题上，我们选择“更深”的模型，更匹配大自然设计的生物结构。

### 5.3.4 深层网络结构

小结一下，数学上证明深层模型训练更快，生物学意义上深层模型匹配生物神经系统结构。一个通用的深层网络模型框架如图 5-35 所示。

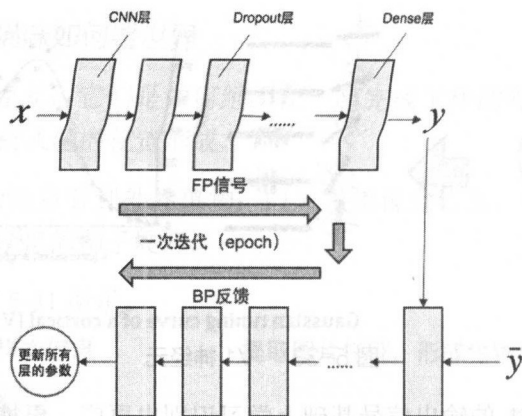


图 5-35 FP-BP 深层网络模型

在某些深度学习类框架中，层的定义并不局限，既可以是 DNN 模型层，也可以是其他神经元模型组成的神经网络层，如后续将要登场的 CNN、RNN 等，还可以是某种数据处理手段，比如 5.4.2 节介绍的 Dropout 层等。

注意，这里依旧使用梯度下降完成模型的训练，在每个梯度迭代中，每一层模型依次逆向更新参数，一次迭代完成了所有层参数的更新。也就是说，虽然层的功能并不相同，但所有层是融合在一起训练的。后层模型按损失函数反馈更新前层模型的参数。

神经元的深层连接模型是本书中深度学习的核心理念之一。

### 5.3.5 回顾

- 我们选择“深层”的方式扩展模型：
  - 数学的链式法则构造出高效的计算流程管道。
  - 深层网络结构匹配生物神经系统的层级特征结构。
- 深层网络的层可以是神经元模型，也可以是数据处理手段。

## 5.4 典型的 DNN 深层网络模型：MLP

想象力比知识更重要。因为知识是有限的，而想象力是无限的，它包含了一切，推动着进步，是人类进化的源泉。

——爱因斯坦

本节将实现 MLP 模型（Multilayer Perceptron，多层感知模型）。

首先介绍一些工程上优化模型训练速度的技巧。

### 5.4.1 优化梯度下降

过去我们一直用梯度下降训练模型，优化模型的参数。梯度下降的执行流程是，每次更新梯度时需要把数据集中每个样本都计算一遍，在海量数据的深度学习中，这个计算量非常大。

#### SGD (Stochastic gradient descent, 随机梯度下降)

一个简单的想法是不再使用全部数据集计算梯度，而是随机抽取一批数据计算（数据量参数：batch\_size），这样计算量会大大减少。直观上，这么做导致的结果就是梯度下降的每个迭代中，并不保证计算出的梯度方向是正确的，我们可能走很多“弯路”。

常规梯度和随机梯度对比如图 5-3 所示。

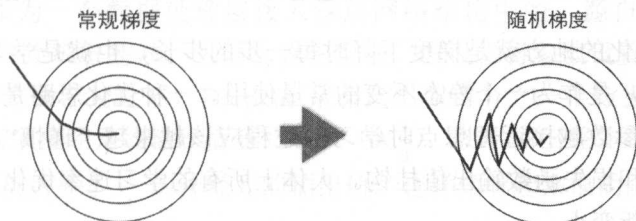


图 5-36 常规梯度 vs 随机梯度（图中的圆圈是等高线）

对于“弯路”问题，我们通过增加迭代次数补偿（iteration），因为对样本的抽取是随机的，所以就算偶尔有抽到错误指向的样本集，如果迭代次数足够多，那么最终也能朝着正确的方向优化参数。

迭代次数的增加对计算机硬件是可以接受的，因为计算机很擅长进行快速串行执行命令。这样的梯度下降算法我们叫作 SGD 算法。

在 SGD 基础上，梯度下降过程中还有两个地方可以优化：

- (1) 通过动量（参数：momentum），优化“弯路”轨迹。
- (2) 优化梯度下降时每一步的步长，或者说学习速率（learning rate，参数：lr）。

关于第一点，对一个常走弯路，但大方向上正确的糊涂蛋，我们可以引入“惯性”的物理概念让他少走弯路。比如缓存一个之前梯度的平均，每次梯度更新时都和之前的平均梯度相加，即  $w_j = w_j + \alpha * (0.9 * \frac{\sum_{i=1}^{j-1} \Delta w_{j-1}}{j-1} + 0.1 * \Delta w_j)$ 。这样每次梯度更新时，新的梯度继承历史梯度的惯性得到一定的纠错能力。在实际工程中，“动量”在梯度的应用往



往往使梯度下降收敛得更快，如图 5-37 所示。

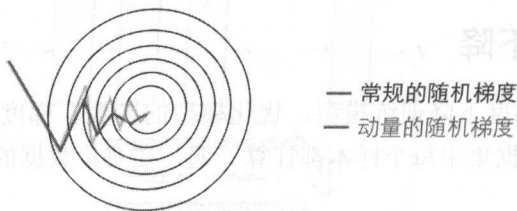


图 5-37 常规随机梯度 vs 动量随机梯度

动量梯度更新：

```
import numpy as np

d_w_list = []
def update_w(w, alpha, d_w_new):
    d_w_list.add(d_w_new)
    return w + alpha * (.9 * np.mean(d_w_list[:-1]) + .1 * d_w_new)
```

另一个可以优化的地方就是梯度下降时每一步的步长，也就是学习速率参数  $lr$ 。之前的梯度下降中， $lr$  是作为一个静态不变的常量使用。一种优化思路是将其指数衰减，内在的直观思想是当参数越接近理想点时学习的过程应该越来越“谨慎”。还有一些思路是将学习速率的大小和损失函数输出值挂钩。大体上所有的学习速率优化方式都是学习速率随着学习的过程不断变小。

随机批量数据 (batch)、梯度动量 momentum 和学习速率参数 ( $lr$ ) 是优化梯度下降的三个重要部分。工程上，常见的优化器如 RMSprop、Adagrad 都是在这三个地方对原始的梯度下降算法动手术，区别只是具体的实现方式不同而已。

在这里，结合第 1 章梯度下降的内容，我们总结一下梯度下降的重要知识点。

- 梯度下降需要输入在同一尺度上可比较。处理手段：对数据及去均值化、归一化，使其满足 0 均值，等方差。
- 随机初始化参数权重，初始化的数值满足 0 均值，等方差。
- 批量训练数据，参数名 batch。
- 动量，参数名 momentum。
- 学习速率，参数  $lr$ 。

虽然梯度下降有很多需要手动调节的参数，但在实际使用中，我们主要关注的还是学习速率。如果成绩表现不佳，你可以先降低学习速率试试，更慢的训练过程往往让模型成绩更佳。

## 5.4.2 处理过拟合：Dropout

在浅层模型中，处理过拟合的常用手段是减少特征、增加数据量或者调试模型的 L2 正则化参数等。在深层模型中，你可以继续使用 L2，也可以选择新的手段：Dropout。如图 5-38 所示。

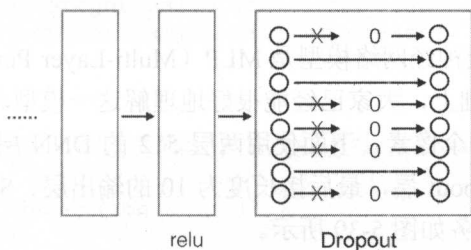


图 5-38 Dropout

Dropout 是作为一个数据处理层接入深层网络结构中的，源自 2012 年大学教授 GEHinton 提出的一个疯狂想法。为什么说它疯狂，假设你将参数设置为 Dropout(0.25)，Dropout 告诉你，在激活层之后，Dropout 层会随机取四分之一数据集的激活值，将其改为 0，同时让剩余神经元的输出值等比放大 ( $\times \frac{4}{3}$ )，使层输出整体维持量级恒定。也就是说，Dropout 随机丢弃了  $\frac{1}{4}$  的神经元的输出值。这样做听起来很胡闹，但在工程上却被证明对过拟合效果惊人的好，为什么呢？

直观地讲，Dropout 告诉模型，你不能依赖任何神经元的激活值，因为他们随时可能被扔掉。这一无赖行为强迫每个神经元负责了部分其他神经元的工作，学习了冗余的表达式。

这就好比说话的时候，一个人说了一堆冗余的话，但往往能让核心意思表达得更清楚。某些个神经元虽然被丢弃了，但总有其他神经元保留了这个神经元携带的信息，让模型最终看起来依然表现得不错。Dropout 强迫模型接受一些随机性，却使系统整体更健壮，防止了过拟合。

如果从模型融合的角度理解，Dropout 的处理结果相当于随机构造了一些不同神经元组成的神经网络模型，并使其融合，利用随机个体形成的群体智慧消除了过拟合。

另外从生物角度讲，Dropout 也再次匹配了生物组织的特性。在生物刺激信号来临时，每个神经元细胞的输出信号其实是一个概率信号，和 Dropout 层的神经元一样，有可能输出，也有可能没音。生物上神经组织使用一群神经元群共同处理输入信号，即使某些个体是不靠谱的，但通过海量个体组织形成不对任何个体过分依赖的群体组织，会使这个生物

系统更健壮。在这一点上，工程设计又意外地匹配了自然的设计！

大自然花了 5 亿多年才进化出类似人类这样如此出色的智能生物结构，相比之下，计算机智能的历史才刚刚开始。

### 5.4.3 MLP 模型

一个典型的多层全连接神经网络模型是 MLP (Multi-Layer Perceptron, 多层感知器神经网络)，相信在上文的基础上，大家已经能很好地理解这一模型。还是手写识别 MNIST 的例子，输入图片向量 784 个像素，下面使用两层 512 的 DNN 层，让神经元数量略大于任务需求，激活后接入 Dropout 层，最后接长度为 10 的输出层、Softmax 概率转换到图片标签向量。MLP-MNIST 任务如图 5-39 所示。

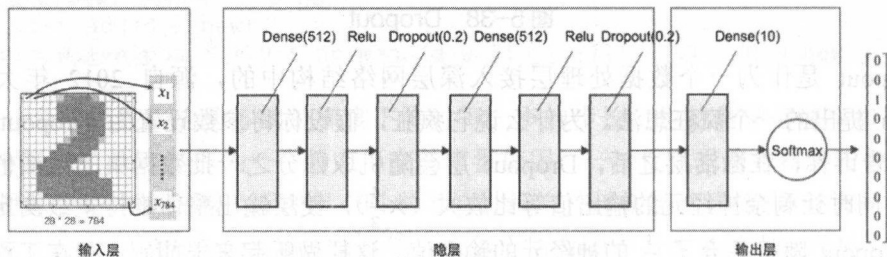


图 5-39 MLP-MNIST 任务

#### Keras 实现 MLP

先看一看 Keras 版本的实现，几行代码，非常简单。

第一步，准备数据：

```
import numpy as np

np.random.seed(1337)
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras.optimizers import SGD, Adam, RMSprop
from keras.utils import np_utils

nb_classes = 10 # 类别
img_size = 28 * 28 # 输入图片大小

# 加载数据，已执行 shuffle-split (训练-测试集随机分割)
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# 以 TensorFlow 为后端，归一化输入数据，生成图片向量
```

```

X_train = X_train.reshape(y_train.shape[0], img_size).astype(
    'float32') / 255
X_test = X_test.reshape(y_test.shape[0], img_size).astype(
    'float32') / 255

# One-Hot 编码标签, 将如[3,2,...] 编码成
# [[0,0,0,1,0,0,0,0,0,0], [0,0,1,0,0,0,0,0,0],...]
Y_train = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)

```

## 第二步, 定义模型:

```

# 创建模型, 增加一层包含 128 个神经元的隐层; Dense 默认 activation 为 linear 线性
model = Sequential([
    Dense(512, input_shape=(img_size,)),
    Activation('relu'),
    Dropout(0.2),
    Dense(512, input_shape=(512,)),
    Activation('relu'),
    Dropout(0.2),
    Dense(10, input_shape=(512,), activation='softmax'),
])

model.summary()

```

## 输出:

Layer (type)	Output Shape	Param #	Connected to
dense_1 (Dense)	(None, 512)	401920	
dense_input_1[0][0]			
activation_1 (Activation)	(None, 512)	0	dense_1[0][0]
dropout_1 (Dropout)	(None, 512)	0	
activation_1[0][0]			
dense_2 (Dense)	(None, 512)	262656	dropout_1[0][0]
activation_2 (Activation)	(None, 512)	0	dense_2[0][0]
dropout_2 (Dropout)	(None, 512)	0	
activation_2[0][0]			
dense_3 (Dense)	(None, 10)	5130	dropout_2[0][0]
=====			
Total params: 669,706			
Trainable params: 669,706			
Non-trainable params: 0			



第三步，编译模型，选择“rmsprop”优化器完成模型的训练：

```
# 编译模型
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

最后，完成剩下的训练及测试环节：

```
batch_size = 128 # 梯度下降批数据量
nb_epoch = 10 # 循环训练集次数

# 训练
model.fit(X_train, Y_train,
          batch_size=batch_size, nb_epoch=nb_epoch,
          verbose=1, validation_data=(X_test, Y_test))

# 测试
score = model.evaluate(X_test, Y_test, verbose=0)
print('accuracy: {}'.format(score[1]))
```

输出：

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [=====] - 13s - loss: 0.2446 - acc:
0.9239 - val_loss: 0.1211 - val_acc: 0.9631
Epoch 2/10
60000/60000 [=====] - 14s - loss: 0.1025 - acc:
0.9688 - val_loss: 0.0796 - val_acc: 0.9750
Epoch 3/10
60000/60000 [=====] - 15s - loss: 0.0758 - acc:
0.9776 - val_loss: 0.0768 - val_acc: 0.9774
Epoch 4/10
60000/60000 [=====] - 15s - loss: 0.0609 - acc:
0.9820 - val_loss: 0.1066 - val_acc: 0.9678
Epoch 5/10
60000/60000 [=====] - 14s - loss: 0.0517 - acc:
0.9845 - val_loss: 0.0878 - val_acc: 0.9782
Epoch 6/10
60000/60000 [=====] - 14s - loss: 0.0441 - acc:
0.9866 - val_loss: 0.0770 - val_acc: 0.9822
Epoch 7/10
60000/60000 [=====] - 14s - loss: 0.0403 - acc:
0.9879 - val_loss: 0.0752 - val_acc: 0.9821
Epoch 8/10
60000/60000 [=====] - 14s - loss: 0.0362 - acc:
0.9892 - val_loss: 0.0769 - val_acc: 0.9831
Epoch 9/10
60000/60000 [=====] - 13s - loss: 0.0310 - acc:
0.9909 - val_loss: 0.0817 - val_acc: 0.9840
```



```
Epoch 10/10
60000/60000 [=====] - 15s - loss: 0.0295 - acc:
0.9914 - val_loss: 0.0842 - val_acc: 0.9827
accuracy: 0.9827
```

最后的成绩约 98% 上下。补充训练参数说明。

- `batch_size` 是梯度下降一个 batch (批) 的样本数量。训练时拿一个 batch 的样本计算一次梯度下降, 更新一次梯度。
- `nb_epoch` 整数, 训练数据将会被遍历 `nb_epoch` 次。(说明: `nb` 指 “number of”)。

注意, 一次 epoch 是指训练集整体被遍历一遍, 根据 `batch_size` 大小决定包含多少个 iteration; 一次 iteration 对应的是一个 batch 数据, 这时梯度参数更新一次。也就是说, 使用 SGD 训练深度学习模型时, 训练集可以被训练很多次。

#### 5.4.4 回顾

本节介绍了 MLP 模型及 Keras 实现。

- 优化梯度下降。
- 随机梯度、批数据 batch。
- 引入动量。
- 变速步长 (学习速率)。
- Dropout 去过拟合。
- Keras 实现 MLP 模型。

### 5.5 Caffe 实现 MLP

本节将使用 Caffe 实现 MLP 模型, 重点介绍训练 Caffe 模型的调式参数。

#### 5.5.1 搭建 MLP

在 Caffe 上实现 MLP 模型, 只需在第 4 章的基础上, 更换模型文件 `train_val.prototxt` 就可以了。

`train_val.prototxt` 文件内容:

```
name: "MLP"
layer {
  name: "data"
  type: "Data"
```

```

top: "data"
top: "label"
include {
    phase: TRAIN # 训练数据层
}
transform_param {
    scale: 0.00392156862745 # 归一化缩小
}
data_param {
    source: "/root/maxmon/lr/gen/train_lmdb" # 训练集地址
    batch_size: 40 # batch 样本数量
    backend: LMDB
}
}
layer {
    name: "data" # 测试数据层
    type: "Data"
    top: "data"
    top: "label"
    include {
        phase: TEST
    }
    transform_param {
        scale: 0.00392156862745 # 归一化缩小
    }
    data_param {
        source: "/root/maxmon/lr/gen/test_lmdb" # 测试集地址
        batch_size: 40 # batch 样本数量
        backend: LMDB
    }
}
}
layer {
    name: "ip1" # 512 DNN 全连接层 (隐层)
    type: "InnerProduct"
    bottom: "data"
    top: "ip1"
    inner_product_param {
        num_output: 512
        weight_filler {
            type: "xavier" # weight 初始化方式
        }
    }
}
}
layer {
    name: "act1" # relu 层
    type: "ReLU"
    bottom: "ip1"
    top: "ip1"
}
}

```

```

layer {
  name: "drop1" # Dropout 层
  type: "Dropout"
  bottom: "ip1"
  top: "ip1"
  dropout_param {
    dropout_ratio: 0.2 # 丢弃比例
  }
}
layer {
  name: "ip2" # DNN 输出层
  type: "InnerProduct"
  bottom: "ip1"
  top: "ip2"
  inner_product_param {
    num_output: 10 # 输出 10 个类别
    weight_filler {
      type: "xavier" # weight 初始化方式
    }
  }
}
layer {
  name: "act2" # ReLU 层
  type: "ReLU"
  bottom: "ip2"
  top: "ip2"
}
layer {
  name: "drop2" # Dropout 层
  type: "Dropout"
  bottom: "ip2"
  top: "ip2"
  dropout_param {
    dropout_ratio: 0.2
  }
}
layer {
  name: "accuracy" # 计算准确率
  type: "Accuracy"
  bottom: "ip2"
  bottom: "label"
  top: "accuracy"
}
layer {
  name: "loss" # softmax 层+loss 损失函数计算层
  type: "SoftmaxWithLoss"
  bottom: "ip2"
  bottom: "label"
  top: "loss"
}

```



和第 4 章相比，这里新增了 ReLu 层（type: ）和 Dropout 层。

## 1. ReLu 层

ReLu 配置：

```
layer {
  name: "act1"
  type: "ReLU"
  bottom: "ip1"
  top: "ip1"
}
```

注意，Caffe 中 ReLu 层的 bottom、top 都是它依附的卷积层。

## 2. Dropout 层

Dropout 配置：

```
layer {
  name: "drop1"
  type: "Dropout"
  bottom: "ip1"
  top: "ip1"
  dropout_param {
    dropout_ratio: 0.2 # 丢弃样本的概率
  }
}
```

和 ReLu 相比，Dropout 层的 bottom、top 都是它依附的卷积层；参数 dropout\_ratio 表示丢弃的几率。

## 3. solver 调试参数

下面详细说明 solver 及相关的参数。这里的 solver.prototxt 文件和第 4 章中的一样：

```
net: "/root/maxmon/lr/pb/train_val.prototxt"
test_iter: 250 # 测试的时候，迭代 250 个 batch
test_interval: 1000 # 每训练迭代 1000 个 batch，测试一次
base_lr: 0.01 # 梯度的基础学习速率
display: 1000 # 每 1000 迭代显示一次当前的成绩
max_iter: 10000 # 最大迭代的次数
lr_policy: "step" # 学习速率递减的方式，配合 gamma、stepsize 使用
gamma: 0.1 # 配合 lr_policy: step
momentum: 0.9 # 动量参数，继承上一次梯度更新的权重
weight_decay: 0.0005 # 权重衰减是参数权重更新规则中的附加项
stepsize: 5000 # 配合 lr_policy: step
```

```
snapshot: 10000 # 每10000次迭代保存训练结果
snapshot_prefix: "/root/maxmon/lr/gen/snapshot_train"
solver_mode: CPU
```

#### 4. test\_interval 和 test\_iter

Caffe 是边训练边测试的，训练过程中，每迭代 1000 个 batch (batch\_size: 40, 相当于训练 40 000 个样本)，测试一次。一次测试至少要包含所有的测试图片，测试样本总共有 10 000, batch 大小为 40, 所以 test\_iter: 250 ( $250 \times 40 = 10\,000$ )。

也就是说，在一个 epoch 里，训练集中的样本要遍历一遍，但测试集的所有样本至少要测试遍历一次，也可能很多次，遍历次数甚至可以不是整数次，关于训练参数你大致了解就可以了。一次完整的训练要经过一个或多个 epoch。

#### 5. 梯度参数

momentum 是梯度下降的动量参数，控制计算当前梯度时继承历史梯度的比例。weight\_decay 是控制权重 weight 更新量衰减的因子。

前面提过，梯度下降的迭代过程中，学习速率合理衰减有益于结果。可以为 lr\_policy 设置不同的值变更学习率，常用的如下。

- fixed: 保持 base\_lr 不变。
- step: step、stepsize、gamma 组合使用。  
梯度更新公式:  $\text{base\_lr} \times \text{gamma}^{\text{floor}(\text{iter} \div \text{stepsize})}$  (iter 是迭代次数)。
- exp: 指数衰减。  
梯度更新公式:  $\text{base\_lr} \times \text{gamma}^{\text{iter}}$  (iter 是迭代次数)。

其余方式有 inv、multistep、poly、sigmoid (和概率转换函数同一个公式，不同使用意义)，感兴趣的读者可以查阅官方文档了解更多。可以看到，使用 Caffe 可以将模型调试得非常细致。

### 5.5.2 训练模型

其余配置文件和脚本请参考 3.4 节，在终端执行“/你的 Caffe 安装目录/build/tools/caffe train --solver pb/solver.prototxt”，模型就可以开始训练了。注意以下几个训练问题。

- 训练时，如果提示测试的误差“loss = nan”，说明权重值太大，需要降低梯度下降的 base\_lr 学习速率。
- 终端 Ctrl+c 中断训练时会自动保存一次，所以 snapshot 大点也没关系。



### 5.5.3 回顾

本章介绍了 DNN 神经元模型组建的深层全连接神经网络，重点解释了本书深度学习的一些核心理念。后续章节将介绍更多其他类型的神经元模型。

# 学习空间特征

生物体通过五感接触外界世界，将其转换成生物神经信号，传递给大脑处理。也就是说，生物体以生物信号描述外界事物，训练大脑模型，如图 6-1 所示。

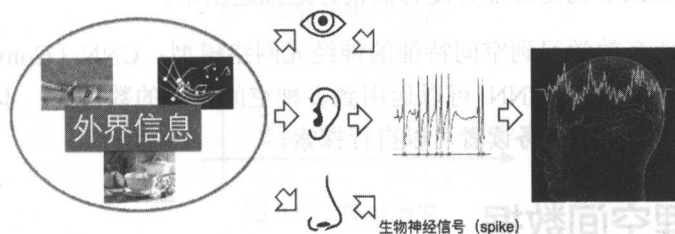


图 6-1 生物接收处理外界信息的方式

计算机用数据描述世界。类似生物的五种感官，外界的事物被编码成像素数组、向量序列等不同的数据形式。这些数据形式接着被转换成特征向量，用于训练模型。也就是说，类比生物脑模型的信息输入形式是生物神经信号，机器学习模型的信息输入形式是特征向量（或者向量数组），如图 6-2 所示。

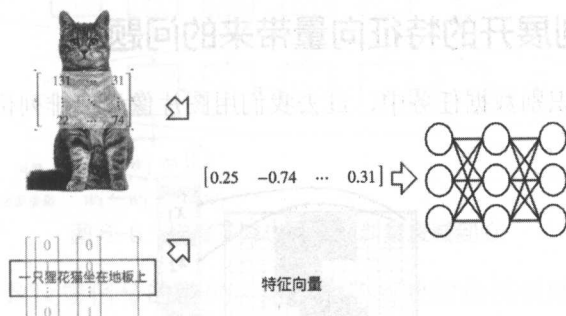


图 6-2 计算机接收处理外界信息的方式

其中一类数据形式是“空间数据”。之所以这么称呼，是因为这些数据通过空间的相对位置关系形成特征。比如图片、像素在平面空间的聚集形成可识别的图形特征，如图 6-3 所示。

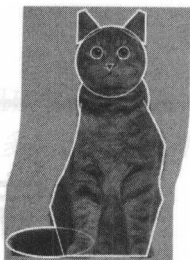


图 6-3 猫的图形特征

(图片出自 TED, Fei-Fei Li: How we're teaching computers to understand pictures)

这些形状特征千变万化，不好描述，所以过去我们直接将图片按像素位置排列依次展开，生成“像素位置”的特征向量。这种展开方式的问题是破坏了一部分图片像素的2D位置关系，并且图形视觉特征并没有被很好地描述出来。

本章将介绍能有效学习到空间特征的神经网络模型：CNN（Convolutional Neuron Network，卷积神经网络）。CNN 可以运用到多种空间意义的数据上，本书只以图片数据为例讲解 CNN，其他数据任务读者可以自行探索。

## 6.1 预处理空间数据

开了好头等于做了一半。

——贺拉斯《书札》

本节以图片数据为例，介绍空间形式的数据的预处理方法。

### 6.1.1 像素排列展开的特征向量带来的问题

在 MNIST 手写识别数据任务中，过去我们用图片像素按排列位置展开的方式生成特征向量，如图 6-4 所示。

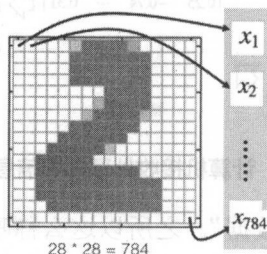


图 6-4 按图片的像素位置展开

这种方式有两个弊端：

- (1) 破坏了一部分图片的 2D 平面结构信息。
- (2) 生成的特征向量对图形特征的描述不够。

特征向量输入到后层的 DNN 模型中，完成 MNIST 分类任务。由于这种方式生成的特征向量质量不佳，因此 DNN 模型需要更多的神经元以及更深的模型层才能从特征向量中解读出分类标签。在一些略复杂的图片识别任务中，这些代价将直接导致我们面临两个问题：(1) 模型的训练非常缓慢；(2) 梯度消失，如图 6-5 所示。

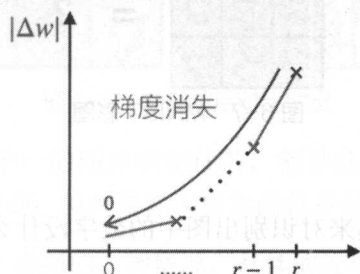


图 6-5 梯度消失

梯度消失是说在模型训练过程中，参数更新的增量沿层从后往前反馈时，由于反馈路径过长，因此在前层的更新量已经缩减为 0——相当于模型实际上没有训练任何东西，如图 6-6 所示。

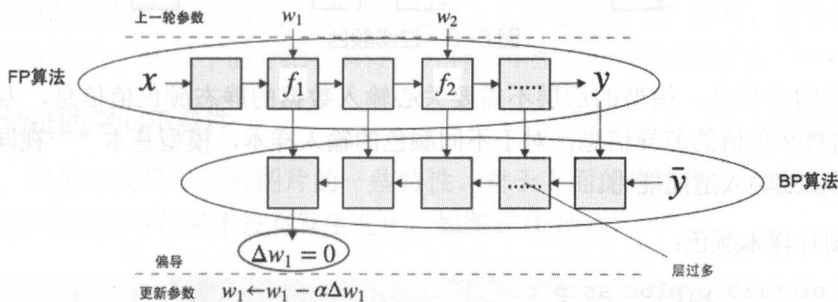


图 6-6 梯度下降过程中更新量衰减到 0

由于距离过远导致反馈信号的消失，这一问题反映的是模型结构本身的瓶颈，目前基本无能为力。所以需要从源头优化模型，下面将从两个方面优化 DNN 模型的输入数据质量，帮助 DNN 模型“减负”：

- (1) 预处理图片数据。
- (2) 生成更合理的图形特征向量。

本节后续部分将介绍第一点：图片数据的预处理手段，即图片样本的过滤及生成；后续小节会介绍第二点。

## 6.1.2 过滤冗余

“过滤冗余”是说，如果输入数据样本里有一些对任务目标没有意义的信息，那么应在一开始就过滤掉这些信息，不让它流入模型。以 MNIST 任务为例，如图 6-7 所示。

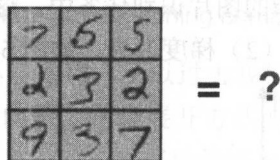


图 6-7 MNIST 彩图

备注：图中数字实际为彩图。

图片样本的颜色信息看起来对识别出图中的数字没什么作用，因此可以过滤掉这部分信息，如图 6-8 所示。

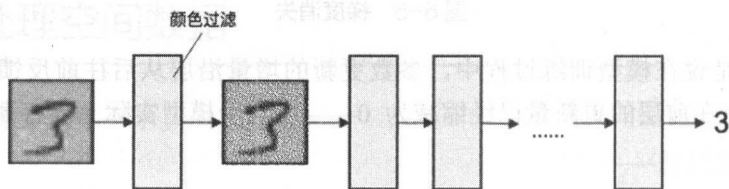


图 6-8 过滤颜色

这样做的好处是，模型的后层不需要关心输入数据的静态颜色值信息，专心学习识别图片中的数字色值的差异信息，对于不同颜色的输入样本，模型基本“一视同仁”——模型有了更强的输入适配能力。

过滤图片样本颜色：

```
import matplotlib.pyplot as plt
from PIL import Image

img_color = Image.open("img/6/mnist_color.png") # 原始色彩图
img_gray = img_color.convert('1') # 转换成黑白图

fig, (ax1, ax2) = plt.subplots(1, 2) # 1 行，每行 2 张 plot，对比

ax1.axis("off") # 关闭坐标轴
ax2.axis("off")
ax1.set_title('colored')
```



```
ax1.imshow(img_color)
ax2.set_title('gray')
ax2.imshow(img_gray)
```

输出如图 6-9 所示。

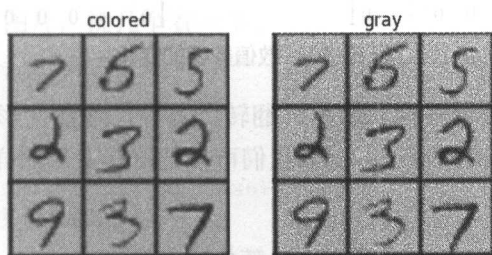


图 6-9 处理前后对比输出

对于一些样本背景色不单一的图片识别任务，额外的颜色信息往往会增加图片中对象实物的辨识度，所以不能过滤；但如果输入样本的背景是纯色，那么就可以过滤掉它。

颜色过滤是“过滤冗余”思路的一种情况，具体的处理方式根据任务而定。

### 6.1.3 生成数据

之前提过，深度学习模型自动学习识别事物特征的能力十分依赖海量的数据样本。在图片识别任务的实际运用中，图片数量不足往往是模型成绩的瓶颈。而另一些识别任务中，样本的获取代价高昂（比如在医疗领域利用人工智能识别 CT 图片，每个正样本意味着一个痛苦的病患）。所以下面介绍在有限图片样本中生成新样本的技巧。

#### 实物特征的空间不变性

首先，我们需要简单了解图片的一些特性。对于下面的样本图片，将图片中的实物“平移”，图中的实物特征基本没有发生变化，如图 6-10 所示。

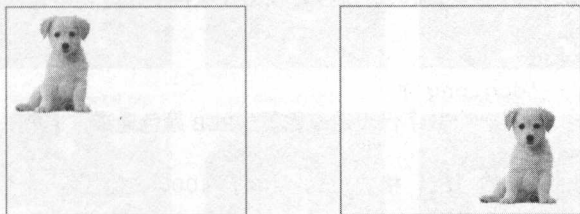


图 6-10 平移图片中实物

在图 6-10 中，“狗”这一实物的特征几乎没变：毛茸茸、扁鼻子……但对于计算机而言，样本的数值数组却变了很多，如图 6-11 所示。

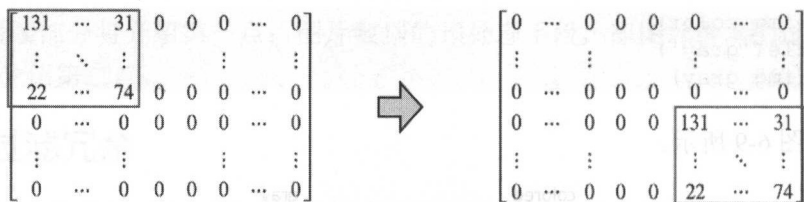


图 6-11 数值数组的变化

其他的图片变形手段有平移、裁剪、翻转，等等，这些变形手段改变了图形，但基本不会破坏图片中对象的实物特征，所以我们可以把这些变形的样本放入训练集，让模型学会“看”出变形的事物。

下面以一张照片样本为例，使用 Keras 随机组合平移、裁剪、翻转等平面处理手段，生成变形的图片样本。

图片变形样本生成。

```
from keras.preprocessing.image import ImageDataGenerator, \
    array_to_img, img_to_array, load_img

# 更多 API 参见官方文档“图片预处理”。https://keras-
cn.readthedocs.io/en/latest/preprocessing/image/""

DIR = '../data/moredata/'

# 随机组合变形手段
pic_gener = ImageDataGenerator(
    rotation_range=20, # 随机旋转角度范围
    width_shift_range=0.1, # 随机水平移动的范围
    height_shift_range=0.2, # 随机垂直移动的范围
    shear_range=0.2, # 裁剪程度
    zoom_range=0.5, # 随机局部放大的程度
    horizontal_flip=True, # 水平翻转
    fill_mode='nearest') # 当旋转、位移导致图片像素空缺时，填充新像素的方式

# 加载成 PIL 图片 400*400
img = load_img(DIR + 'dog.png')
# 转换成 (400, 400, 3)，第一个数字代表通道数量 (RGB 颜色通道: 3 个)
x = img_to_array(img)
# 转换成 Keras/TensorFlow 的“tf”格式 (1, 400, 400, 3)，第一个数字代表图片数量
x = x.reshape((1,) + x.shape)

i = 0
# 生成随机“变形”的图片；这里可以用 flow 对单张图片进行处理，
# 也可以用 flow_from_directory 对文件夹内所有图片进行处理
```

```
for batch in pic_gener.flow(x, batch_size=1,
                             save_to_dir=DIR[:-1], save_prefix='gen',
                             save_format='png'):
    i += 1
    if i > 30: break # 生成 30 张
```

上面生成了 30 张新图片，挑几个看一下。

```
import matplotlib.pyplot as plt

# 2 行，每行 3 张 plot
fig, ((ax1, ax2, ax3), (ax4, ax5, ax6)) = plt.subplots(2, 3)
for ax in (ax1, ax2, ax3, ax4, ax5, ax6):
    ax.axis("off") # 关闭坐标轴

# 看几张生成的图片
TITLE_SIZE = 8
ax1.set_title('original', fontsize=TITLE_SIZE)
ax1.imshow(img)
ax2.set_title('slope-crop', fontsize=TITLE_SIZE)
ax2.imshow(load_img(DIR + 'gen_0_2403.png'))
ax3.set_title('partial_enlarged', fontsize=TITLE_SIZE)
ax3.imshow(load_img(DIR + 'gen_0_3380.png'))
ax4.set_title('crop-horizontal_flip', fontsize=TITLE_SIZE)
ax4.imshow(load_img(DIR + 'gen_0_448.png'))
ax5.set_title('slope-crop2', fontsize=TITLE_SIZE)
ax5.imshow(load_img(DIR + 'gen_0_3582.png'))
ax6.set_title('slope-crop-horizontal_flip', fontsize=TITLE_SIZE)
ax6.imshow(load_img(DIR + 'gen_0_4009.png'))
plt.show()
```

输出如图 6-12 所示。

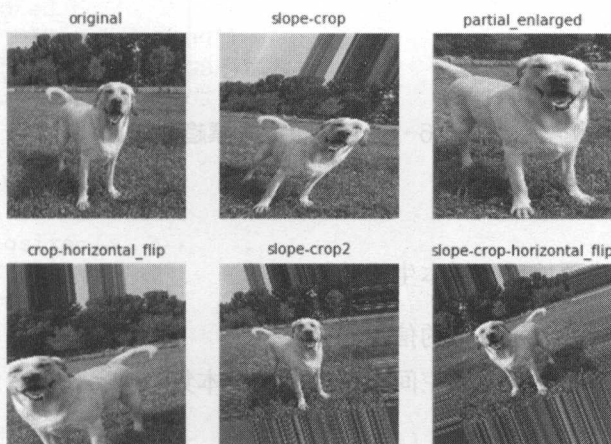


图 6-12 变形后的图片

用肉眼来看还是那条狗，但对于计算机而言已经是像素值大不相同的数据样本了。图片虽然变形了，但图片中事物的实物特征基本没有受影响。

除了 Keras 的 API 实现图片变形，你还可以使用 OpenCV 完成更复杂的图像变形，由于篇幅有限，这里不再介绍。

值得注意的是，并不是所有图片识别任务都适合用变形的方式增加样本。

之前狗狗的图片中，图片变形并不会破坏待识别事物的特征，而对于如图 6-13 所示的股票 K 线图识别，反转、裁剪手段会直接破坏图中事物的特征意义，比如反转直接将一个股价上升特征变为了下降，这样增加的样本只会造成干扰。所以在运用技术解决问题的时候，我们要具体问题具体分析。

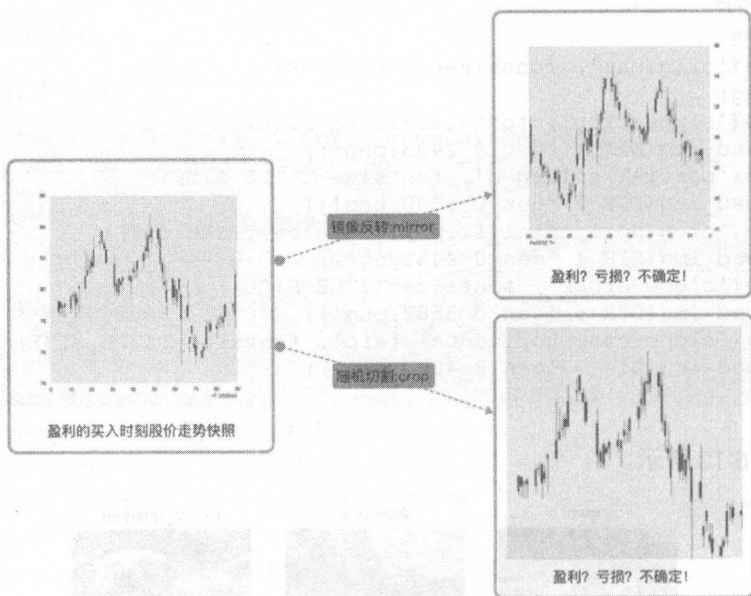


图 6-13 变形后的股票趋势图

#### 6.1.4 回顾

本节介绍了图片预处理及样本生成的技巧：

- 过滤对于目标识别冗余的信息。
- 利用图片中实物特征的空间不变性扩大样本集。

## 6.2 描述图片的空间特征：特征图

生活中从不缺少美，而是缺少发现美的眼睛。

——罗丹

如何生成能够更合理描述图片样本特征的向量，首先需要能够提取空间特征的运算方式，这就是卷积运算。

### 6.2.1 图片的卷积运算

“卷积”是什么？从本质上讲，卷积和加减乘除一样，不过是一种定义好的运算法则。

$$(f * g)[n] \stackrel{\text{def}}{=} \sum_{m=-\infty}^{\infty} f[m]g[n-m]$$

备注：上面是一维数据的卷积公式。

从定义上看，卷积运算规则包含两部分： $f$  和  $g$ 。将  $f$  视为输入， $g$  作为卷积核，卷积核  $g$  对输入的  $f$  执行卷积操作。卷积是一种线性运算，那么它的意义是什么呢？

对于工程师而言，其实不需要特别理解这种数学公式本身的含义，这些只是数学家设计的游戏规则，我们应该更关注运算结果带来的应用意义。下面将一张输入图片作为  $f$ ：

```
import cv2 # OpenCV 是一个非常棒的视觉处理库，cv2 是 OpenCV 的 Python 版
```

```
DIR = '../data/moredata/'
img = cv2.imread(DIR + 'dog.png')
# cv2 用 BGR 格式编码图片，转换成 RGB 存储
f = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
```

```
def show_img(img):
    """展示图片"""
    plt.figure(figsize=(3, 3))
    plt.axis('off')
    plt.imshow(img)
    plt.show()
```

```
show_img(f)
print('展示一部分图片像素值:\n' + str(f[:2, :2]))
```

输出如图 6-14 所示。





图 6-14 狗照

代码如下。

展示一部分图片像素值：

```
[[[179 199 239]
  [178 198 240]]

 [[190 205 237]
  [186 203 238]]]
```

同样，将一个二维数组作为卷积核  $g$ ：

```
g = np.array([[[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], dtype='float32')
```

卷积运算之后，得到一个二维数值数组，下面将它作为图片画出来看看：

```
# 执行卷积运算，也就是 cv2 的 filter2D。命名 filter 源自信号滤波的概念，本节后面会
# 提到
cov = cv2.filter2D(f, -1, g)

show_img(cov)
```

输出如图 6-15 所示。



图 6-15 卷积后的结果

这就是特征图。从结果上看，特征图就像是原始图片经过特殊处理后生产的图片数组！这个效果具体是怎么得到的呢？下面稍微展开一些卷积的内部运算细节，以方便解释。

## 平移和叠加

图片数据和卷积核的卷积运算可以拆成两个物理概念理解：

- (1) 卷积中心的平移。
- (2) 卷积核的加权叠加。

卷积中心的平移是指卷积运算的像素中心从左向右，从上向下依次移动。卷积核的加权叠加是指在每个卷积中心代表的像素位置上，以周围像素按卷积核参数为权值，叠加在一起生成新的中心像素。

比如在图 6-16 中，卷积核在计算中心（在像素值为 252 的地方），那么当前的计算输出值为  $-1 \times 93 + 0 \times 139 + 1 \times 101 - 2 \times 26 + 0 \times 252 + 2 \times 196 - 1 \times 135 + 0 \times 230 + 1 \times 18 = 231$ ，即新生成的中心点输出值为 231，对应旧的中心点输入值 252。这样就完成了一个点值的卷积运算。然后卷积中心扫描过所有输入像素点，这样就生成了一张“特征图”。

131	162	232	84	91	207
104	-1	0	+1	237	109
243	-2	0	+2	135	26
185	-1	0	+1	61	225
157	124	25	14	102	108
5	155	176	218	232	249

图 6-16 在图片二维数组上卷积运算

## 6.2.2 卷积指令和特征图

那么特征图有什么用呢？

这一点的答案藏在卷积核的参数中。因为卷积运算实际上就是让图片像素按卷积核参数加权叠加，生成新的像素，那么卷积核的参数表示的应用意义就是描述以什么样的方式生成新图的指令。

比如我们希望生成的特征图和原来一样，这一指令以卷积像素的方式可以翻译为“中心像素为 1，周围像素权值为 0（不叠加周围像素）”，那么对应卷积核参数的数字化

0 0 0  
 的语言描述为:  $\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$ 。  
 0 0 0

```
img_bgr = cv2.imread(DIR + 'dog_1.png') # 原始图片
# cv2 用 BGR 格式编码图片, 这行代码会转换成 RGB 存储
origin = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)
kernel_do_nothing = np.array([[0, 0, 0], [0, 1, 0], [0, 0, 0]], \
dtype='float32') # 用 float 类型计算
```

这样生成的特征图和原图一致:

```
def compare_imgs(imgs, titles=[]):
    """对比图片"""
    n = len(imgs)
    titles = titles if titles else [''] * n # 没传初始化为等长的空 list
    fig, axes = plt.subplots(1, n, figsize=(12, 12))
    for ax, img, title in zip(axes, imgs, titles):
        ax.axis("off") # 关闭坐标轴
        ax.imshow(img)
        ax.set_title(title, fontsize=12)
    plt.show()

cov = cv2.filter2D(origin, -1, kernel_do_nothing) # 执行卷积运算
compare_imgs([origin, cov], ['original', 'conversionnal'])
```

输出如图 6-17 所示。

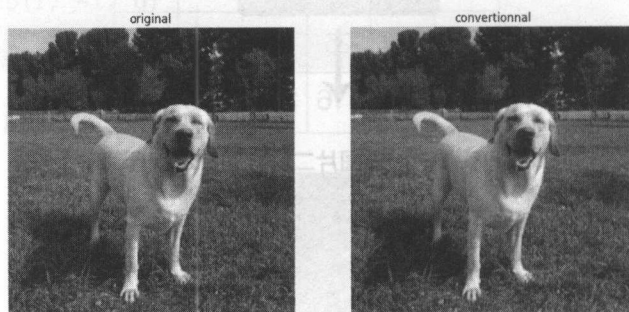


图 6-17 原图和卷积图对比

如果期望特征图描述原图的“整体模糊印象”，那么这个目标可以通过“让每个中心像素和周围的像素等比例叠加”这一方式实现，也就是图片的均值模糊处理。对应的卷积

核可以设计成:  $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \div \text{sum}$ 。

```
g = np.ones((3, 3), dtype='float32') # 均值模糊
kernel_mean = g / np.sum(g) # 让卷积核数组归一化，所有参数加起来等于1
```

让卷积核数组归一化的意义在于保持新生成的像素值和原来的值在同一范围，这样特征图的亮度不会发生变化。

对比两张图：

```
cov = cv2.filter2D(origin, -1, kernel_mean) # 执行卷积运算
compare_imgs([origin, cov], ['original', 'cov_mean'])
```

输出如图 6-18 所示。

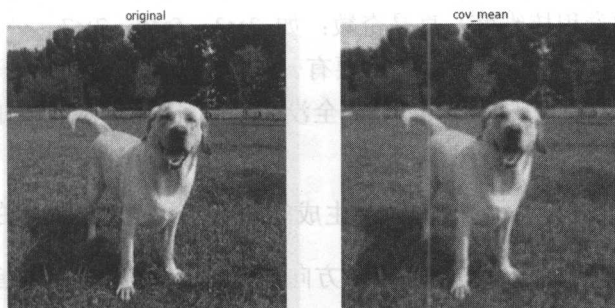


图 6-18 原图和特征图对比

可以让指令的作用半径扩大，换言之告诉中心点“叠加更广的像素区域”——卷积

核数组变为  $5 \times 5$  的  $\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \div \text{sum}$ ，甚至更大，观察效果：

```
g3 = np.ones((3, 3), dtype='float32') # 均值模糊
kernel_mean3 = g3 / np.sum(g3) # 让卷积核数组归一化，使所有参数加起来等于1

g7 = np.ones((7, 7), dtype='float32')
kernel_mean7 = g7 / np.sum(g7)

g13 = np.ones((13, 13), dtype='float32')
kernel_mean13 = g13 / np.sum(g13)

cov3 = cv2.filter2D(origin, -1, kernel_mean3) # 执行卷积运算
cov7 = cv2.filter2D(origin, -1, kernel_mean7)
cov13 = cv2.filter2D(origin, -1, kernel_mean13)
compare_imgs([origin, cov3, cov7, cov13], ['original', 'cov_mean_3x3',
'cov_mean_7x7', 'cov_mean_13x13'])
```

输出如图 6-19 所示。

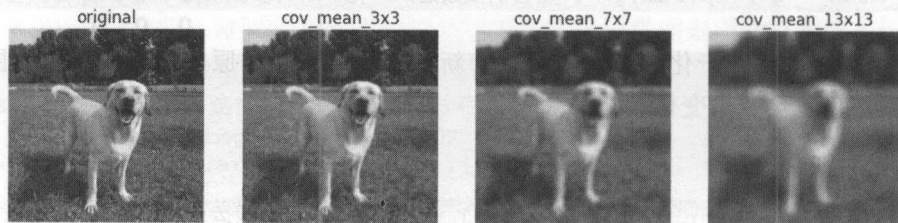


图 6-19 多个卷积后的特征图对比

特征图越来越模糊的原因是卷积核叠加的像素范围越来越广。

注意，上面的卷积核维度一直是奇数：如  $3 \times 3$ 、 $5 \times 5$ 、 $7 \times 7$ 、…。奇数的原因在于二维图片的卷积运算平移和叠加时，始终要有一个图片意义上的“卷积中心点”。当然，你可以按照心情随意尝试偶数卷积核，这完全没有问题。但在机器学习的模型实验中，每一步都应该是可解释的。

接下来尝试更多的指令。比如希望生成的特征图描述原图样本的“运动模糊”效果，像素的实现方式就是“卷积中心只朝一个方向叠加像素”，数字化的指令是 
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \div 3$$
 或者同等形式的高阶数组。

```
g5 = np.identity(5, dtype='float32') # 运动模糊
kernel_move5 = g5 / np.sum(g5)
g11 = np.identity(11, dtype='float32') # 高阶卷积核上看效果更明显
kernel_move11 = g11 / np.sum(g11)

cov5 = cv2.filter2D(origin, -1, kernel_move5) # 执行卷积运算
cov11 = cv2.filter2D(origin, -1, kernel_move11)
compare_imgs([origin, cov5, cov11], ['original', 'cov_move_5x5',
'cov_move_11x11'])
```

输出如图 6-20 所示。



图 6-20 原图和运动模糊特征图对比



下面尝试一些更明显描述原图样本“特征”意义的特征图，比如“轮廓”特征。像素的实现方式是“凸显中心像素和周围像素的对比”，卷积核设计为

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

```
# 轮廓卷积核；同时放大2倍，让凸显效果更清楚一点
kernel_contour = np.array([[ -1, -1, -1], [-1, 8, -1], [-1, -1, -1]],
dtype='float32') * 2

cov = cv2.filter2D(origin, -1, kernel_contour) # 执行卷积运算
compare_imgs([origin, cov], ['original', 'cov_contour'])
```

输出如图6-21所示。

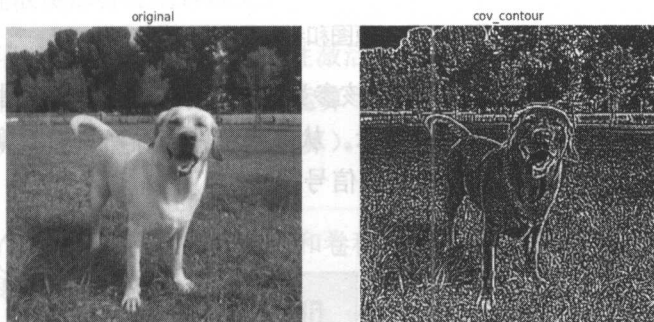


图6-21 原图和轮廓特征图对比

正如你所看到的那样，上面按正负数分配中心像素和周围像素的权值参数，同时使卷积核数组的参数加起来为0。也就是说，如果中心像素和周围像素数值一样，则这个像素的输出值为0（黑暗），反之越亮。这样的卷积核过滤了图片像素不变的部分，凸显了差异的部分，因此就提取到了原图样本的“轮廓”特征。

还有很多其他类型的特征图，比如边缘  $\begin{bmatrix} 1 & 1 & 1 \\ 1 & -7 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ 、浮雕  $\begin{bmatrix} -1 & -1 & 0 \\ -1 & 3 & 0 \\ 0 & 0 & 0 \end{bmatrix}$  等。

```
# 边缘卷积核，参数和加起来大于1，图片会更亮一些。像素指令：加强周边，抑制自身
kernel_edge = np.array([[1, 1, 1], [1, -7, 1], [1, 1, 1]],
dtype='float32')
```

```
# 浮雕卷积核，一种3D阴影效果
```

```
# 指令是将中心点一边的像素减去中心或者另一边的像素；倍数代表放大的程度
```

```
kernel_carve = np.array([[ -1, -1, 0], [-1, 3, 0], [0, 0, 0]],
dtype='float32') * 3
```

```
cov_edge = cv2.filter2D(origin, -1, kernel_edge) # 执行卷积运算
cov_carve = cv2.filter2D(origin, -1, kernel_carve)
```

```
compare_imgs([origin, cov_edge, cov_carve],
             ['original', 'cov_edge', 'cov_carve'])
```

输出如图 6-22 所示。

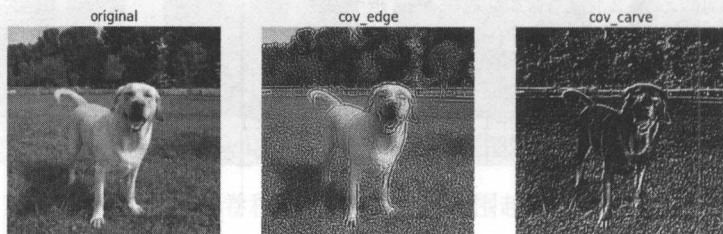


图 6-22 原图和卷积特征图对比

总的来说，卷积运算对图片按卷积核参数加权叠加，生成特征图。在信号领域这一过程又叫作滤波（filter），如图 6-23 所示。从信号角度来看，卷积运算的结果过滤了输入的一部分信号，凸显了与输入相关的特征信号。

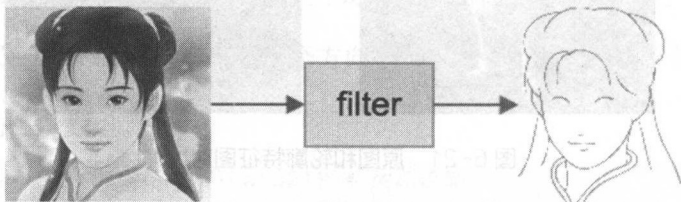


图 6-23 轮廓滤波

所以这里卷积核的作用就相当于“滤波器（filter）”，即描述过滤信号的方式。

### 6.2.3 回顾

- 图片的卷积运算。
  - 卷积中心的平移。
  - 中心周围像素的叠加。
- 卷积核就是提取特征“指令”，卷积运算生成“特征图”。这个过程在信号领域称之为“滤波”，卷积核就是“滤波器”。

## 6.3 CNN 模型 I：卷积神经网络原理

因为它就该是这样！这件事把我以往生活中的疑问奇妙地串了起来，太神奇

了……就仿佛我突然间读懂了自己的命运。

——《塔希里亚故事集》

卷积运算能够提取图片的空间特征的一部分进行描述，本节我们将利用这一特性，将卷积作为神经元的内核运算，设计出可以提取样本空间特征的模型：CNN 模型（Convolutional Neural Network，卷积神经网络）。

### 6.3.1 卷积神经元

最终的目标是设计出能够将图片样本转换成合理特征向量的模型层。根据卷积运算，我们首先设计能够提取图形特征的神经元。

神经元的结构依旧是“线性内核+非线性激活”（参见 5.2 节）。计算机数据是离散数据，而离散卷积运算在数学上被证明是线性运算（就像 6.2 节你看到的那样，在离散图片数据上，卷积运算就是像素值的系数乘法运算）。如果将卷积运算作为内核，那么这就是卷积神经元。

在图片任务中，对比一下 DNN 神经元和卷积神经元，两者的主要区别就是内核零件的更换和数据的维度变化，如图 6-24 所示。

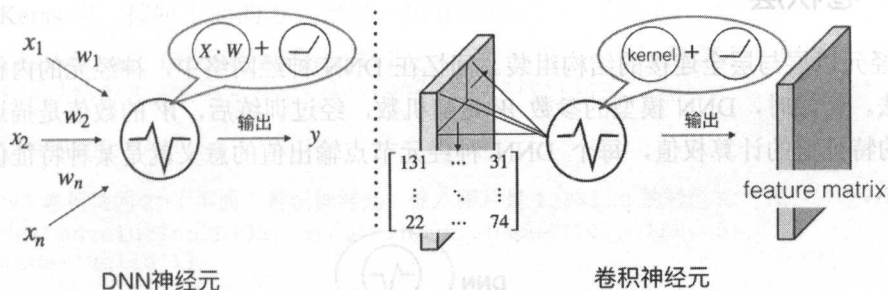


图 6-24 DNN 神经元和卷积神经元输入/输出对比

- (1) 输入样本方面：卷积神经元输入从 DNN 神经元的特征向量变成了二维图片数组。
- (2) 训练的参数方面：DNN 神经元是矩阵系数相乘，训练权值为  $W$ ；卷积神经元则是卷积运算，训练卷积核的参数（二维数组）。
- (3) 输出方面：对于输入样本，DNN 神经元输出它在某种特征下的数值，而卷积神经元则输出它的某种特征图。
- (4) 模型层结构方面：双方都会后接一个 ReLu 层对神经元进行激活。
- (5) 从特征意义上说，在目标分类任务中，DNN 神经元识别了某种“类别特征”，

而卷积神经元则识别了“空间特征”。

上一节中，我们人工设计了几个卷积核参数，发现它们的卷积运算有提取图形特征的作用。而深度学习的主题一直是去掉人工的特征设计，让模型从数据集中自动发现特征。所以在卷积神经元中，我们放弃了人工指定的卷积核，让模型通过训练（梯度下降）自动发现合理的卷积核参数。

这里补充拓展一下理解，从机器学习模型训练的角度来看，卷积运算的加权叠加性质，就是以空间的方式关联待训练的参数  $W$ 。如图 6-25 所示。

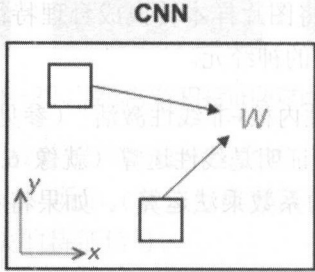


图 6-25 空间关联训练参数

### 6.3.2 卷积层

神经元以层与层全连接的结构组装。回忆在 DNN 神经网络中，神经元的内核运算是矩阵乘法。初始时，DNN 模型的参数  $W$  是随机数，经过训练后， $W$  的数值是描述模型自己发现的特征值的计算权值，每个 DNN 神经元节点输出值的意义就是某种特征值，如图 6-26 所示。

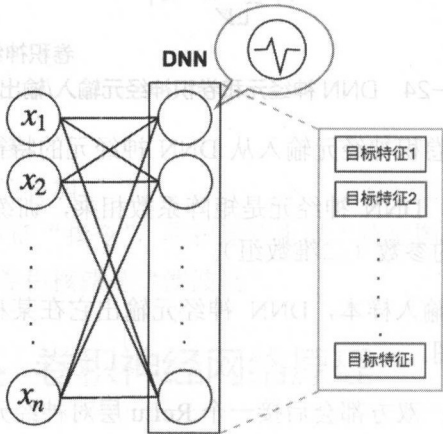


图 6-26 DNN 神经元的输出值的意义（ $W$ 值）

同样地，我们可以组装多个卷积神经元形成卷积层，让卷积层通过模型的训练自动发掘数据集的平面特征。一个卷积神经元扫描输入图片生成一张特征图，所以一层如果有  $N$  个卷积神经元，就输出  $N$  张特征图，对应描述  $N$  种平面特征，如图 6-27 所示。

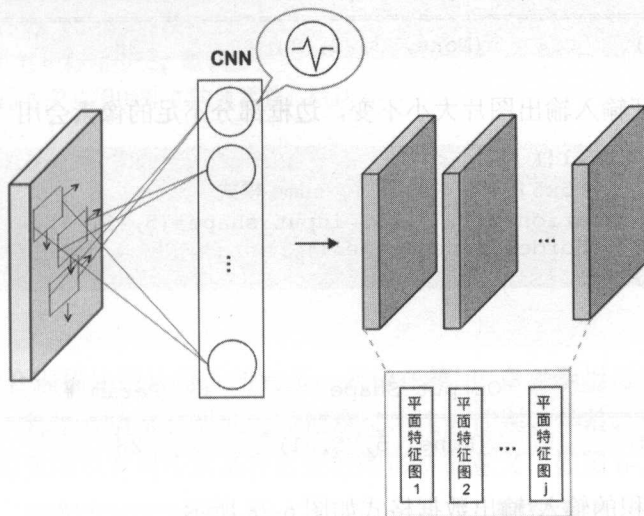


图 6-27 CNN 神经元的输出值意义

在 Keras 中，按照下面的方式定义一层卷积层：

```
from keras.models import Sequential
from keras.layers import Convolution2D, Activation

model = Sequential()

# 32 个 3×3 卷积核的 2D (平面) 卷积神经元；输入图片是 128×128 的彩色 RGB 图片
model.add(Convolution2D(32, 3, 3, input_shape=(128, 128, 3),
border_mode='valid'))

# 或者按照下面的方式，这两种完全等价
# model.add(Convolution2D(32, 3, 3, input_shape=(128, 128, 3)))
```

在 Keras 中，卷积核的卷积运算有两种模式：“valid”（默认）和“same”。卷积运算过程包含两部分：中心像素的平移和周围像素的按核参数叠加。

“valid”是说每次只移动一步，比如输入  $5 \times 5$  的图片，我们用  $3 \times 3$  的卷积核“valid”模式下输出图片的大小是  $(5-3+1)$ ，也就是  $3 \times 3$ 。

```
tmp_model = Sequential()
# 1 个 3×3 卷积核；输入是 5×5 的彩色 RGB 图片；valid 模式
tmp_model.add(Convolution2D(1, 3, 3, input_shape=(5, 5, 3),
border_mode='valid'))
```



```
tmp_model.summary()
```

输出:

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 3, 3, 1)	28

“same” 则保证输入输出图片大小不变，边框部分不足的像素会用“0”填充:

```
tmp_model = Sequential()
# 1 个 3x3 卷积核; 输入是 5x5 的彩色 RGB 图片; same 模式
tmp_model.add(Convolution2D(1, 3, 3, input_shape=(5, 5, 3),
                                border_mode='same'))
tmp_model.summary()
```

输出:

Layer (type)	Output Shape	Param #
conv2d_4 (Conv2D)	(None, 5, 5, 1)	28

不同模式下卷积的输入/输出数据格式如图 6-28 所示。

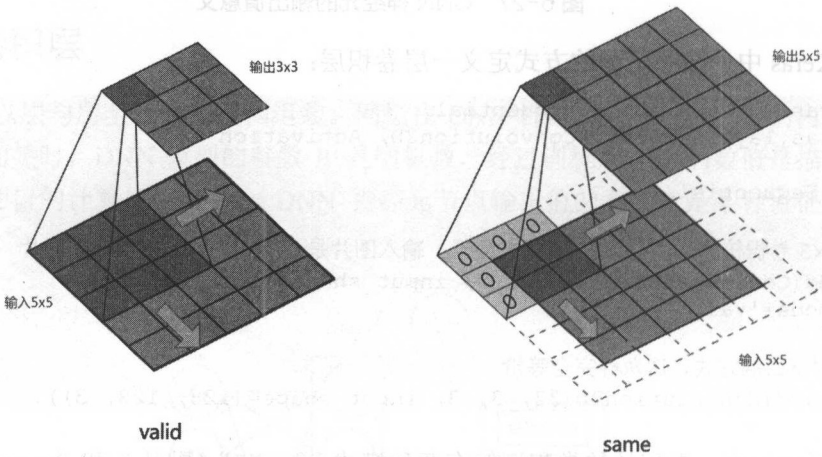


图 6-28 不同模式下卷积的输入/输出数据格式

Keras 选择将最常用的卷积模式封装成 `border_mode`，但 Caffe 要自由得多，我们可以选择 `same` 模式 (`stride: 1`)，也可以随意设定卷积时一次移动几个格子 (`stride: 任意合理范围的数字`)。

Caffe 使用配置文件描述模型，一个典型的卷积层的配置如下:

```
layer {
  name: "conv"
```

```

type: "Convolution"
bottom: "data"
top: "conv1"
convolution_param {
  num_output: 32 # 输出维度/卷积核数量
  kernel_size: 3 # 3×3 卷积核
  stride: 1 # 卷积移动步长, 默认 1
  pad: 0 # 在输入图片周围补 0 的像素数, 默认 0
  weight_filler {
    type: "xavier" # 初始化 weight 的方式
  }
  bias_filler {
    type: "constant" # 初始化 bias 的方式
  }
}
}

```

目标是获得合理描述图片特征的向量。就像在第 1 章“泰坦尼克号生存预测”任务的样本向量一样，每个数值都有对应的特征意义，比如性别、年龄。而现在，我们将特征图排列成向量，每张图也有对应的某种合理的图形特征意义，如图 6-29 所示。

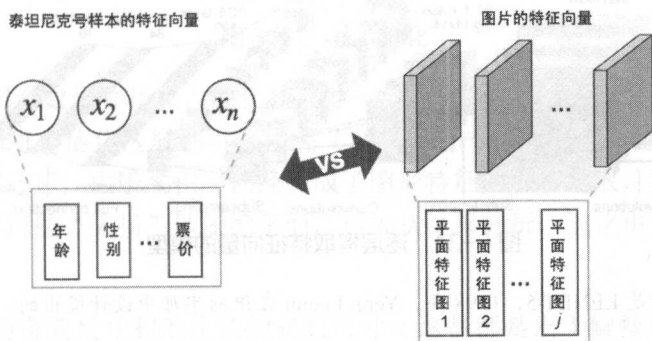


图 6-29 机器学习的特征向量和 CNN 输出的平面特征图组向量对比

接下来，需要把这个“特征图向量”压缩成特征向量，就像一个高质量的 MP3 一样，我们希望压缩尽量无损。

### 6.3.3 多层卷积

在生物系统中，当我们把事物从一种形式变换成另一种形式时，是以逐层渐变的方式完成的。比如在生物脑中“看”到的事物，其实是每层神经元提取分析一些特征，逐层抽象，最后形成整体影像，如图 6-30 所示。

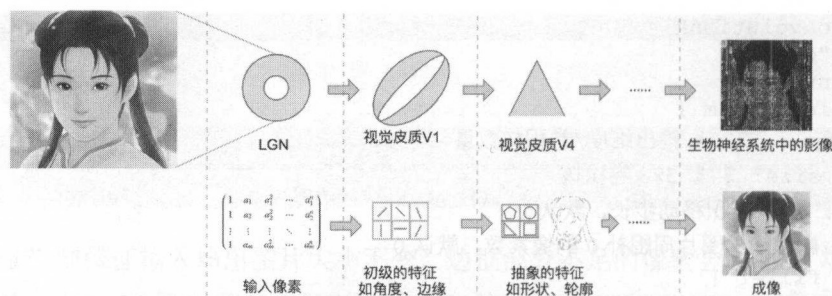


图 6-30 逐层抽象的生物认知

这里将模仿生物，具体实现是这样的：添加多个卷积层，每个卷积层压缩图片的维度，对应拉升向量的长度，以逐层渐变的方式将特征图向量拉伸成特征向量的形式。

如图 6-31 所示，每层缩小图片的大小对应拉升向量长度，直至展开形成向量（S4→C5），最后的“Flatten”层按像素位置展开向量，这时样本大部分的图形特征已经能很好地提取。

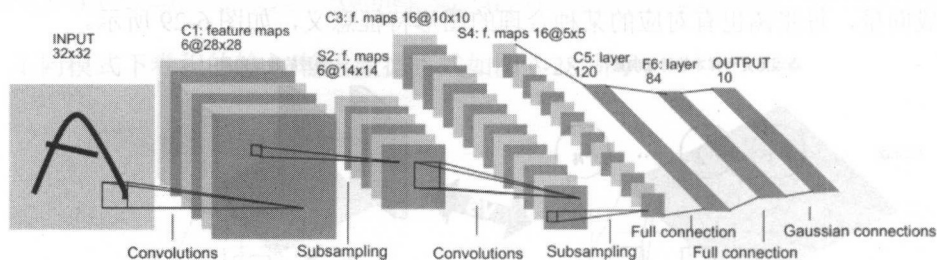


图 6-31 逐层提取特征向量的模型

备注：上图模型是 LENET-5，1998 年，Yenn Lecun 在识别字母中设计提出的。

从描述事物的特征角度来理解，这种网络结构的意义在于逐步过滤掉图片样本的平面结构特征，保留其中描述实物的特征部分，也就是最后输出的特征向量。

这是整体结构，下面说明如何压缩图片平面尺寸。

## 池化

上面的网络结构需要逐步压缩平面尺寸，并对应的拉升向量长度，所以我们关心如何实现这两点：平面的缩小和特征向量维度的增长。

特征维度增长好做，就是权衡任务需求和计算性能，然后逐层放置更多的卷积神经元。至于平面的缩小，有两种方式：

(1) 让卷积运算的卷积核移动步长大于 1，即跳跃式平移。这样输出的尺寸会等比例

缩小。

## (2) 池化 (pooling)。

第一种方式其实割裂了一些平面信息，不常用，所以这里主要介绍池化。

以最大池化 (max-pooling) 为例。我们指定一个类似于卷积核的池化核，这个核不携带参数，而是携带某种操作函数，比如最大函数，输出输入像素中的最大值。接着，将这个池化核在输入图片的上下左右平移，如图 6-32 所示。

### MAX POOLING

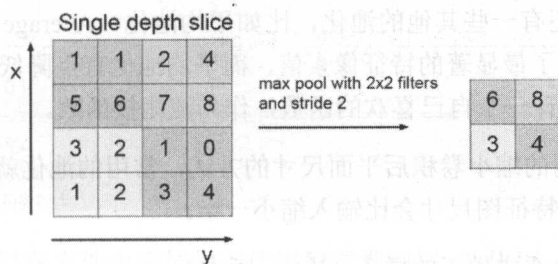


图 6-32 最大池化

和卷积核常用奇数尺寸相反，池化核尺寸一般是偶数，并且每次移动的步长是 2 的倍数。一般而言，每个池化区域都不再重叠（现在也有一些模型应用重叠的池化手段）。池化层放在卷积层之后，由于卷积层已经完成了图形特征的合理提取，而池化的设计初衷只是在图形区域尽可能“无损”地压缩平面，后来从结果分析，池化也表现出一些“聚合”特征的效果。

关于卷积核的奇数尺寸和池化核的偶数尺寸，这些都是出于解释意义的常规——由于模型每一步的构造、调试都应该是可解释的，所以一般都会这么设定，但不是必须的。

在 Keras 中，用如下代码增加池化层：

```
model.add(MaxPooling2D(pool_size=(2, 2)))
```

在 Caffe 中，用如下代码配置一个池化层：

```
layer {
  name: "pool"
  type: "Pooling"
  bottom: "conv"
  top: "pool1"
  param {
    lr_mult: 1 # 卷积 weight 的学习速率倍数 (基础数值是 base_lr)
    decay_mult: 1 # weight 学习速率衰减因子
```



```

}
param {
  lr_mult: 2 # bias 的学习速率倍数一般为 weight 的两倍 (实验结果给出的建议)
  decay_mult: 0 # bias 学习速率衰减因子
}
pooling_param {
  pool: MAX # 指定最大池化。可选最大池化、均值池化、随机池化
  kernel_size: 2 # 2×2 池化核
  stride: 2 # 池化步长为 2, 默认为 1
  pad: 0 # 在输入图片周围补 0 的像素数, 默认 0
}

```

除了最大池化, 还有一些其他的池化, 比如平均池化 (Average Pooling)。从图像意义上看, 最大池化突出了最显著的特征像素值, 而平均池化有点降低整体图片分辨率的意思。当然, 你也可以设计一个自己喜欢的函数, 作为池化核函数。

池化是现在最常用的缩小卷积后平面尺寸的方式, 常用的池化就是 2×2, 步长为 2 的最大池化, 这样输出的特征图尺寸会比输入缩小一半。

现在可以将模型改造成如下的样子 (Keras 版本):

```

from keras.layers import MaxPooling2D, Dropout, Flatten

# 套路: 卷积-激活-池化
model = Sequential([
    Convolution2D(32, 3, 3, input_shape=(128, 128, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Convolution2D(64, 3, 3, activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(), # 按像素位置平铺展开
])

# 下面和上面的定义方式等价
# model = Sequential()
# model.add(Convolution2D(32, 3, 3, input_shape=(128, 128, 3),
# activation='relu'))
# model.add(MaxPooling2D(pool_size=(2, 2)))
# model.add(Convolution2D(64, 3, 3, activation='relu'))
# model.add(MaxPooling2D(pool_size=(2, 2)))
# model.add(Flatten())

```

整体模型如下所示, 注意每层输入/输出的格式:

```
model.summary()
```

输出:



Layer (type)	Output Shape	Param #	Connected to
convolution2d_1 (Convolution2D)	(None, 126, 126, 32)	896	
convolution2d_input_1[0][0]			
maxpooling2d_1 (MaxPooling2D)	(None, 63, 63, 32)	0	
convolution2d_1[0][0]			
convolution2d_2 (Convolution2D)	(None, 61, 61, 64)	18496	
maxpooling2d_1[0][0]			
maxpooling2d_2 (MaxPooling2D)	(None, 30, 30, 64)	0	
convolution2d_2[0][0]			
flatten_1 (Flatten)	(None, 57600)	0	
maxpooling2d_2[0][0]			
Total params: 19,392			
Trainable params: 19,392			
Non-trainable params: 0			

在 Caffe 中，可以在文件中添加如下代码配置卷积层+池化层：

```
layer {
  name: "conv" # 32 3x3 卷积层
  type: "Convolution"
  bottom: "data"
  top: "conv"
  param {
    lr_mult: 1 # 梯度下降 weight 参数步长变化倍数
  }
  param {
    lr_mult: 2 # 梯度下降 bias 参数步长变化倍数，一般为 weight 的 2 倍
  }
  convolution_param {
    num_output: 32 # 32 个卷积核
    kernel_size: 3 # 3x3 卷积核
    stride: 1 # 步长 1
    weight_filler {
      type: "xavier" # weight 初始化方式
    }
    bias_filler {
      type: "constant" # bias 初始化方式
    }
  }
}

layer {
  name: "pool" # 池化层
```

```

type: "Pooling"
bottom: "conv"
top: "pool"
pooling_param {
  pool: MAX # 最大池化
  kernel_size: 2 # 2x2 池化
  stride: 2 # 步长 2
}
}

```

总结一下，我们以多层“卷积层+池化层”的方式提取样本的图形特征，生成特征向量。从特征意义上理解，多层 CNN 的每层都在提取样本的图形特征，并努力将这些特征映射到向量上。对比前后层可以发现，前层的图形特征需要较大的图片描述，而后层的图形特征大部分已经充分提取并映射到向量上。

拓展理解：从滤波的角度看，CNN 模型过滤掉了样本的平面结构信号，保留了特征信号。

### 6.3.4 回顾

本节我们介绍了卷积神经网络模型的原理。6.4 节将以一个任务例子，完成模型的训练。

- 卷积神经元：卷积内核以空间的方式关联待训练的参数  $W$ 。
- 卷积层：提取生成“特征图向量”。
- 多层卷积+池化：逐层压缩平面，提取图形特征描述并映射到向量上。从滤波角度看，过滤掉平面结构信号，保留特征信号。

## 6.4 CNN 模型 II：图片识别

Man's Reach Exceeds His Imagination.

——电影《致命魔术》

6.3 节构建了生成特征向量的 CNN 模块，本节我们将结合 DNN 分类模块，使用 Keras 完成图片识别任务。

### 6.4.1 连接分类模型

回顾之前构造的 CNN 模块：

```

from keras.layers import MaxPooling2D, Dropout, Flatten
from keras.models import Sequential

```

```
from keras.layers import Convolution2D, Activation

# 套路：卷积-激活-池化
model = Sequential([
    Convolution2D(32, 3, 3, input_shape=(128, 128, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Convolution2D(64, 3, 3, activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
])
```

将 CNN 模块输出的描述样本的空间图形事物的特征向量，输入到分类模型中，这样就构造出一个完整的图片识别模型，如图 6-33 所示。这个分类模型可以是 DNN 模型，也可以是其他浅层模型，如逻辑分类、随机森林等。

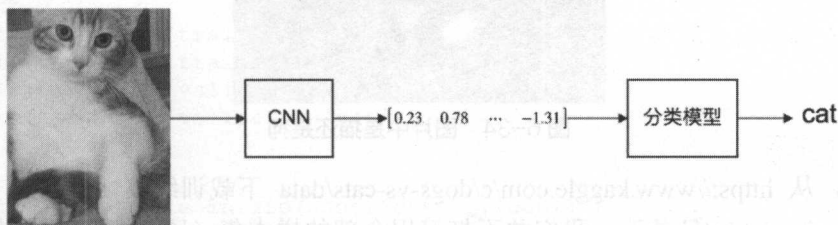


图 6-33 接入分类模型

本书以连接 DNN 为例，代码实现如下：

```
from keras.layers import Dense

# 追加 DNN 层
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))
```

上面我们用了两层 DNN，其中，第一层 DNN 神经元数量设置为 64。这个数值是出于解释意义的设置，因为 CNN 最后一层识别了 64 种空间图形特征，所以第一层 DNN 的神经元应该不少于这个数值。同样，这是出于模型可解释意义的参数选择，不是必需的。

由于下面将使用模型完成二分类的图片识别任务，所以这里模型最后的输出维度只有一维：0-1；并且用了 sigmoid 作为最后的概率-标签转换函数。对于多类别分类任务，修改最后的 Dense 输出维度，并且切换成 softmax 函数转换输出值就可以了。

## 6.4.2 猫狗分类

在第 5 章的 MNIST 例子中，模型的成绩已经接近 100%。所以本章将换一个更具挑战的例子，展示新模型的效果。

Kaggle 是一个很棒的机器学习任务平台，有丰富的学习资源以及优秀的 team。下面从 Kaggle 发布的任务中挑选一个适合本章主题的任务：Dogs VS Cats（猫狗分类）。猫狗分类的数据集是从网络爬取的若干猫或者狗的图片，目标是识别中图片的动物是猫还是狗。

描述任务：二分类问题，给定一张图片，辨别图片中的猫狗，如图 6-34 所示。



图 6-34 图片中是猫还是狗

首先，从 <https://www.kaggle.com/c/dogs-vs-cats/data> 下载训练集 train.zip，将其解压到 ../data/dog\_cat/train/ 目录下。我们并不打算用全部的样本集（因为模型训练速度在单机上不可接受），所以只从训练集中抽取一部分样本作为实验数据。

观察整体 train 训练集：

```
import os
import shutil
import random

train = '../data/dog_cat/train/' # 下载的训练集

dogs = [train + i for i in os.listdir(train) if 'dog' in i]
cats = [train + i for i in os.listdir(train) if 'cat' in i]
print len(dogs), len(cats)
```

输出：

```
12500 12500
```

训练集的标签是按文件名给出的，比如 dog.131.jpg。共 12 500 张猫和狗的图片——类别数量均衡。这个级别的样本数量在单机上训练模型有点慢，所以我们只抽取一部分数据集作为演示样本。下面的代码随机采样了猫狗各 1000 张图片做训练，各 500 张做成绩测试，并将样本按类别建立文件夹，方便模型自动生成标签。



准备数据集:

```
target = '../data/dog_cat/arrange/' # 目标训练集地址

# 随机化
random.shuffle(dogs)
random.shuffle(cats)

def ensure_dir(dir_path):
    if not os.path.exists(dir_path):
        try:
            os.makedirs(dir_path)
        except OSError:
            pass

# 生成文件夹
ensure_dir(target + 'train/dog')
ensure_dir(target + 'train/cat')
ensure_dir(target + 'validation/dog')
ensure_dir(target + 'validation/cat')

# 复制图片
for dog_file, cat_file in zip(dogs, cats)[:1000]:
    shutil.copyfile(dog_file, target + 'train/dog/' +
os.path.basename(dog_file))
    shutil.copyfile(cat_file, target + 'train/cat/' +
os.path.basename(cat_file))

for dog_file, cat_file in zip(dogs, cats)[1000:1500]:
    shutil.copyfile(dog_file, target + 'validation/dog/' +
os.path.basename(dog_file))
    shutil.copyfile(cat_file, target + 'validation/cat/' +
os.path.basename(cat_file))
```

数据集准备完毕, 接下来预处理样本。回忆在 6.1 节中提过的丰富图片样本数量的技巧, 同样地, 这里将用一些图片变形手段, 将原始样本变形, 在提升数据丰富度的同时, 通过训练学习变形事物让模型提取出更抽象的事物特征。

生成变形图片样本:

```
from keras.preprocessing.image import ImageDataGenerator

# 图片尺寸
img_width, img_height = 128, 128
input_shape = (img_width, img_height, 3)

train_data_dir = target + 'train'
validation_data_dir = target + 'validation'

# 生成变形图片
```



```

train_pic_gen = ImageDataGenerator(
    rescale=1. / 255, # 对输入图片归一化到 0-1 区间
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.5,
    horizontal_flip=True, # 水平翻转
    fill_mode='nearest')

# 测试集不做变形处理, 只需要归一化。
# 关于为什么要做归一化, 参见 1.4 节
validation_pic_gen = ImageDataGenerator(rescale=1. / 255)

```

接下来生成数据流。注意, 这里是二分类问题, 标签非 0 即 1, 所以不需要用 `np_utils.to_categorical` 对样本标签进行 One-Hot 编码:

```

# 按文件夹生成训练集流和标签, binary: 二分类
train_flow = train_pic_gen.flow_from_directory(
    train_data_dir,
    target_size=(img_width, img_height),
    batch_size=32,
    class_mode='binary')

# 按文件夹生成测试集流和标签
validation_flow = validation_pic_gen.flow_from_directory(
    validation_data_dir,
    target_size=(img_width, img_height),
    batch_size=32,
    class_mode='binary')

```

输出:

```

Found 3832 images belonging to 2 classes.
Found 1963 images belonging to 2 classes.

```

准备训练模型, 和上面定义的模型基本一致:

```

from keras.models import Sequential
from keras.layers import Convolution2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten, Dense

nb_train_samples = 2000
nb_validation_samples = 1000
nb_epoch = 50 # 循环 50 轮

# 两层卷积-池化, 提取 64 个平面特征
model = Sequential([
    Convolution2D(32, 3, 3, input_shape=input_shape,

```

```

        activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Convolution2D(64, 3, 3, activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
    Dense(64, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid'),
1)

```

# 损失函数设置为二分类交叉熵

```

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

```

数据流的训练 API 和之前的稍不一样:

```

model.fit_generator(
    train_flow,
    samples_per_epoch=nb_train_samples,
    nb_epoch=nb_epoch,
    validation_data=validation_flow,
    nb_val_samples=nb_validation_samples)

```

准确率接近 80%，没有显卡时训练一个 epoch 约 1 分多钟，好的显卡视任务情况能带来几倍甚至几十倍的提速。

注意保存训练结果:

```

ensure_dir(target + 'weights')
model.save_weights(target + 'weights/' + '1.h5')

```

### 6.4.3 反思 CNN 与 DNN 的结合：融合训练

说明：这里的融合训练和第 1 章的模型融合 Ensemble 没有关系。

对于上面的图片识别模型，如果我们按照功能去理解模型模块，则 CNN 负责在特征空间描述样本，DNN 负责识别出样本类别，它们是两个不同功能的模块。然而值得思考的是，在每轮训练迭代中，对于每个样本，CNN 模型层和 DNN 模型层是一起更新梯度参数的，如图 6-35 所示。

一起训练意味着除了 CNN 解读的信息能够通过 FP 传递给 DNN，DNN 计算出的梯度更新也能通过 BP 的方式反馈给 CNN。简单地说，训练时 DNN 的参数可以通过“反馈”影响 CNN 的参数。所以，CNN 模块和 DNN 模块融合的图片识别模型，本质上反映了一个理念：对于同一个任务目标，可以以“层”的方式将不同功能模块融合在一起训练，让它们彼此能够通信，共享信息。

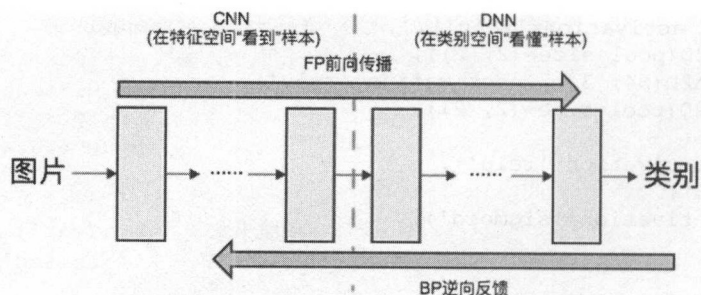


图 6-35 CNN 和 DNN 一起进行梯度训练

过去，一个复杂机器学习任务往往通过几个机器学习模型串联在一起完成。比如对于某个任务，需要模型 A 完成一个机器学习任务，B 在 A 的输出上再完成另一个机器学习任务，模型的最后成绩计算是： $\text{Score}(A) \times \text{Score}(B) \times \dots$

如图 6-36 所示，模型之间彼此独立，每个损失一点准确率，往往就会导致最后的成绩不乐观。而现在，则是通过深度学习的层融合，将若干不同功能的任务模块融合在一起，形成一个模型。由于模型的成绩不再是若干项乘积，并且不同功能模块在训练过程中相互帮助，通过通信反馈共享训练信息，使模型的最终成绩可以有不错的提升——这个理念就是融合训练。近几年，很多复杂任务模型成绩的提升均受益于这一理念。

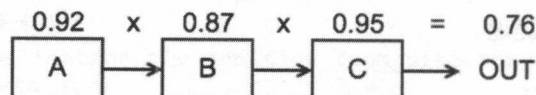


图 6-36 多个机器学习任务串联

融合训练其实是一个宽泛又略抽象的设计理念，贯穿于模型设计的宏观至微观。在 CNN 和 DNN 之间，是“层”的融合，因为这些层有着共同的识别目标。而卷积对参数的叠加计算，可以视为像素/神经元的融合，因为只有图片空间中的像素组合在一起，才形成事物。这就是融合训练的理念：如果两个参数/层指向同一目标，那么我们就把它们放在一起训练，共享彼此。

#### 6.4.4 深度学习与生物视觉

从深度学习模型的角度可以很好地解释一些生物视觉现象。

##### 1. 视幻觉补全

生物上，脑皮层视觉组织将图片用生物信号“成像”，让我们“看到”事物；而另一部分，如语言皮层组织将样本识别成单词描述的事物，让我们“看懂”事物。类似 CNN 和 DNN 的情况，脑模型也是不同脑组织一起训练，识别事物。所以训练模型时，DNN 通

过反馈更新 CNN 的参数，用生物的话讲就是：你“看懂”的事物可以影响你“看到”什么样的事物！（What you think you see, decide what you see!）。

这就解释了一些视觉问题，比如图 6-37 是一个旋转视觉图。

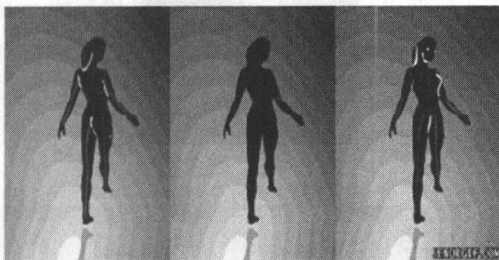


图 6-37 舞者——视错觉（出自果壳网 <http://www.guokr.com/question/483009>）

备注：这是一张动态图，请用浏览器打开原址观看或者查阅随书 Git 库中本章目录下的文件。

这个实验是耶鲁大学耗时 5 年的研究成果。你先看完左图再看中间的，会发现她在顺时针旋转；先看完右图再看中间的，会发现她在逆时针旋转。想想看，为什么？

中间的图其实缺失了部分 2D 信息，而大脑在“看”的时候会自动尝试补全它，关键在于如何补全。先看左图，当你沿着思维的惯性认为她是从左向右转的，视觉就会按从左向右的方式补全信息。反之先看右图，视觉会按从右向左的方式补全。这个实验在一定程度上说明大脑的意识可以影响视觉成像。

## 2. 看到“梦”

如果你看过《盗梦空间》就知道，我们睡觉时大多数的梦境在醒来的一刻都会忘记，梦境对于人类而言似乎是一个无法掌控的黑盒。然而 Jack Gallant 在 2011 年就读 Berkeley 博士学位时做过一个非常魔幻的实验，打破了这个黑盒的神秘，实验旨在解读人类大脑的图片信号，进而记录保存下人类睡眠时丢失的梦境。

实验人员让志愿者头戴 fMRI（记录脑波生物信号的仪器），同时播放一些不同电影，通过 fMRI 存储下来人类视觉皮层的信号数据，然后通过机器学习的技术，重塑 1 秒电影画面数据如图 6-38 所示。图 6-39 是电影图片（上层）和重塑图片（下层）的对比。

现代计算机设备仅仅能重塑 1s 的数据信号，并且有很大的误差和失真，但相信通过不断改进这些仪器和算法，我们最终能迈进那些神秘的领域。





图 6-38 概念图



图 6-39 重塑皮层视觉信号实验（出自 Coursera: Computational Neuron Science）

### 6.4.5 回顾

本节我们完成了一个自设计的 CNN+DNN 图片识别模型。

- CNN 提取图片的特征描述向量，DNN 在此基础上完成识别任务。
- 融合训练。

## 6.5 CNN 的实现模型

6.4 节我们构造了一种 CNN 模型。由于卷积、池化有很多种配置方式，所以可以衍生出很多实现模型。本节将介绍几个被广泛认可的 CNN 系图片识别模型，主要展示模型的网络结构，模型具体如何使用将在后续文章中介绍。

关于这些模型，简单了解就可以了，很多模型的组合方式、配置参数都是大量实验后择优的结果，强行解释意义不大。

### 6.5.1 ImageNet 简介

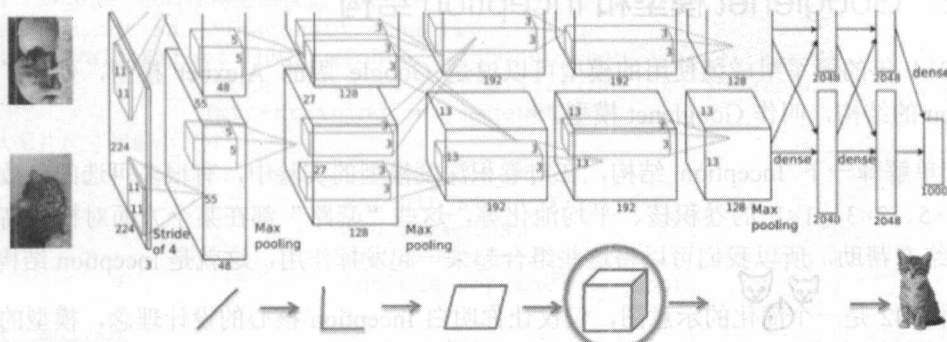
本节介绍的是在 ImageNet 比赛中成绩优秀的 CNN 实现模型。



深度学习模型的成功依赖海量的高质量数据，ImageNet 就是为这一使命诞生的项目。它由斯坦福大学的李飞飞教授领导缔造，是目前世界上图像识别最大的开放数据库。

斯坦福大学每年都会举办一个比赛，邀请谷歌、微软、百度等 IT 企业使用 ImageNet 测试他们的系统运行情况。你可以把这个比赛当作计算机视觉界的世界杯，每年一度的比赛牵动着各大巨头公司的心弦。过去几年中，系统的图像识别功能大大提高，出错率仅约为 5%（比人眼还低）。很多成功的模型借 ImageNet 孵化出来，并且公布给业界，推动了人工智能领域的发展。

在 2012 年，当时的冠军参赛小组设计的卷积模型 ALEXNET，以 16.4% 的错误率（远低于第二名的 26.2%）在 ImageNet 的物体识别比赛中脱颖而出（见图 6-40），从此卷积神经模型才登上舞台，从一个备受质疑的晦涩模型一跃成为了计算机视觉界的香饽饽，几乎当时所有新发表的计算机视觉论文都把它作为核心模型，在 2013 年和 2014 年的挑战赛中，几乎所有的参赛者都采用了卷积神经网络模型。



When AlexNet is processing an image, this is what is happening at each layer.

图 6-40 Alexnet (2012 年 ImageNet 的 ILSVRC 冠军模型)

ImageNet 拥有近 10 亿张图片数据，来自 167 个国家，近 5 万个工作者在一起工作，完成图片筛选、排序、标记。这就是为了支持深度学习模型，ImageNet 付出的精力。

ImageNet 的下一步就是图像理解，项目名称“视觉基因组 (Visual Genome) 计划”，就是把语义和图像结合起来，让机器像三岁的孩子那样，看懂图片并生成句子。项目模型的设计理念上类似在 6.4 节末尾提到的语言层、视觉层、识别层结合在一起的融合模型。最新的进展报告中，李飞飞教授她们已经成功创造了人类历史上第一个“计算机视觉模型”（这里的视觉不是指“看到”，而是指“看懂”）——它在看到图片的第一时间，就有能力生成类似人类语言的句子，如图 6-41 所示。

从 2015 年到本书编写的这段时间，ImageNet 的图片识别比赛前几名主要在模型的

ensemble 上做了很多努力（关于 ensemble 参见 2.6 节），模型本身的突破创新并不如以前那么大。所以接下来将主要介绍 2014 年比赛的冠亚军模型：Inception 结构的 Googlenet 模型和 VGG 模型。

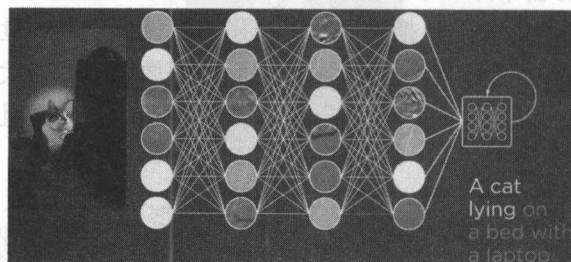


图 6-41 计算机视觉模型——看图说话

备注：出自 TED，Fei-Fei Li: How we're teaching computers to understand pictures。

## 6.5.2 Googlenet 模型和 Inception 结构

2014 年的冠军组成绩使用的模型可以说是 Google 版的 Alexnet 模型，使用了名为 Inception 的结构，叫作 Googlenet 模型。

简单解释一下 Inception 结构，在对卷积深度模型的实验中，有很多可选的参数方案，比如  $5 \times 5$ 、 $3 \times 3$ 、 $1 \times 1$  的卷积核、平均池化等，这些“套路”都在某个方面对构建高效的神经网络有帮助，所以我们可以将这些组合起来一起发挥作用，这就是 Inception 结构。

图 6-42 是一个简化的示意图，仅仅让你明白 Inception 核心的设计理念，模型的具体建造细节是根据大量实验不断调整完成的。

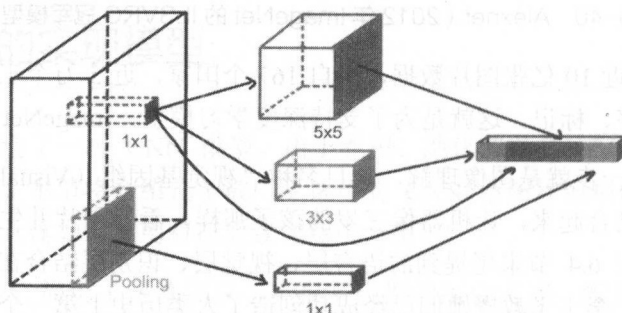


图 6-42 Inception 结构

值得一提的是，其中的  $1 \times 1$  卷积核：卷积运算的意义在于平移和叠加，但在  $1 \times 1$  卷积核上并不存在周围像素的叠加运算。 $1 \times 1$  卷积核的神经元工程意义是，在没改变多少原图信息的基础上，让特征图维度加深（特征向量维度），并在生成的特征图中引入非线性

的因素（结合非线性激活层）。

## 1. Keras 实现 Inception 结构的模型

在 Keras 的 `applications` 包中已经有定义好的 InceptionV3 模型。不同于之前用的 Sequential 模型容器，`applications` 的模型都是用 Model 模型作为容器的，Sequential 模型可以看作 Model 模型的 Quick Start。

Keras 的 `applications` 包里的模型都可以直接选择加载在 ImageNet 数据集上训练好的模型参数，相当于我们可以直接使用在高性能集群上训练过海量数据的成熟模型！

大致浏览一下 `applications` 包中 InceptionV3 模型的定义。

```
def InceptionV3(include_top=True, weights='imagenet',
               input_tensor=None, input_shape=None):

    # 选择 ImageNet 或者使用空白模型
    if weights not in {'imagenet', None}:
        raise ValueError('The `weights` argument should be either '
                          '`None` (random initialization) or `imagenet` '
                          '`(pre-training on ImageNet).`')

    # 默认图片尺寸和最小尺寸
    input_shape = _obtain_input_shape(input_shape,
                                      default_size=299,
                                      min_size=139,
                                      dim_ordering=K.image_dim_ordering(),
                                      include_top=include_top)

    # 省略一部分代码

    x = conv2d_bn(img_input, 32, 3, 3, subsample=(2, 2),
                  border_mode='valid')
    x = conv2d_bn(x, 32, 3, 3, border_mode='valid')
    x = conv2d_bn(x, 64, 3, 3)
    x = MaxPooling2D((3, 3), strides=(2, 2))(x)

    x = conv2d_bn(x, 80, 1, 1, border_mode='valid')
    x = conv2d_bn(x, 192, 3, 3, border_mode='valid')
    x = MaxPooling2D((3, 3), strides=(2, 2))(x)

    # mixed 0, 1, 2: 35 x 35 x 256
    for i in range(3):
        branch1x1 = conv2d_bn(x, 64, 1, 1)

        branch5x5 = conv2d_bn(x, 48, 1, 1)
        branch5x5 = conv2d_bn(branch5x5, 64, 5, 5)

        branch3x3dbl = conv2d_bn(x, 64, 1, 1)
        branch3x3dbl = conv2d_bn(branch3x3dbl, 96, 3, 3)
```

```

branch3x3dbl = conv2d_bn(branch3x3dbl, 96, 3, 3)

branch_pool = AveragePooling2D(
    (3, 3), strides=(1, 1), border_mode='same')(x)
branch_pool = conv2d_bn(branch_pool, 32, 1, 1)
x = merge([branch1x1, branch5x5, branch3x3dbl, branch_pool],
          mode='concat', concat_axis=channel_axis,
          name='mixed' + str(i))

# 省略一大部分代码

if include_top:
    # Classification block
    x = AveragePooling2D((8, 8), strides=(8, 8), name='avg_pool')(x)
    x = Flatten(name='flatten')(x)
    x = Dense(1000, activation='softmax', name='predictions')(x)

# Ensure that the model takes into account
# any potential predecessors of `input_tensor`.
if input_tensor is not None:
    inputs = get_source_inputs(input_tensor)
else:
    inputs = img_input
# Create model.
model = Model(inputs, x, name='inception_v3')

# load weights 加载 ImageNet 训练后的参数
if weights == 'imagenet':
    # 省略代码
return model

```

可以用下面的方式加载定义好的 VGG 模型：

```

from keras.applications.inception_v3 import InceptionV3

base_model = InceptionV3()

```

## 2. Caffe 实现 Googlenet 模型

在你的 Caffe 安装目录/models/bvlc\_googlenet 下 train\_val.prototxt 文件中，可以看到 Inception 结构的模型，Caffe 中习惯将此模型叫作 Googlenet。

完整的 Googlenet 配置文件非常庞大，篇幅有限，这里不再展示。下一章将用 Caffe 完成一个完整的 Googlenet 模型任务。

### 6.5.3 VGG 模型

VGGNet 是 2014 年 ImageNet 图片识别比赛的第二名，VGG 是典型的卷积神经网络



设计，理念上就是卷积+深层模型，然后 deeper and deeper。

图 6-43 就是一个典型的 Keras 的 VGG-16（核心 16 层）模型。



图 6-43 VGG-16 模型

## 1. Keras 实现 VGG 16 模型

Keras 的 applications 收录了两种 VGG 模型，VGG-16（核心 16 层）和 VGG-19（核心 19 层），在不同的机器学习类库中，沿着 VGGNet 思路实现的定义有一些微小差异，我们可以忽略这些差异。

下面浏览一下 Keras 里实现的 VGG-16 的模型定义。

```
def VGG16(include_top=True, weights='imagenet',
          input_tensor=None, input_shape=None):
```



# 省略一部分代码

```
# Block 1
x = Convolution2D(64, 3, 3, activation='relu', border_mode='same',
                  name='block1_conv1')(img_input)
x = Convolution2D(64, 3, 3, activation='relu', border_mode='same',
                  name='block1_conv2')(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='block1_pool')(x)

# Block 2
x = Convolution2D(128, 3, 3, activation='relu', border_mode='same',
                  name='block2_conv1')(x)
x = Convolution2D(128, 3, 3, activation='relu', border_mode='same',
                  name='block2_conv2')(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='block2_pool')(x)

# Block 3
x = Convolution2D(256, 3, 3, activation='relu', border_mode='same',
                  name='block3_conv1')(x)
x = Convolution2D(256, 3, 3, activation='relu', border_mode='same',
                  name='block3_conv2')(x)
x = Convolution2D(256, 3, 3, activation='relu', border_mode='same',
                  name='block3_conv3')(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='block3_pool')(x)

# Block 4
x = Convolution2D(512, 3, 3, activation='relu', border_mode='same',
                  name='block4_conv1')(x)
x = Convolution2D(512, 3, 3, activation='relu', border_mode='same',
                  name='block4_conv2')(x)
x = Convolution2D(512, 3, 3, activation='relu', border_mode='same',
                  name='block4_conv3')(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='block4_pool')(x)

# Block 5
x = Convolution2D(512, 3, 3, activation='relu', border_mode='same',
                  name='block5_conv1')(x)
x = Convolution2D(512, 3, 3, activation='relu', border_mode='same',
                  name='block5_conv2')(x)
x = Convolution2D(512, 3, 3, activation='relu', border_mode='same',
                  name='block5_conv3')(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='block5_pool')(x)

# 是否包括识别层模型
if include_top:
    # Classification block
    x = Flatten(name='flatten')(x)
    x = Dense(4096, activation='relu', name='fc1')(x)
    x = Dense(4096, activation='relu', name='fc2')(x)
    # 输出层, 共 1000 种类别
    x = Dense(1000, activation='softmax', name='predictions')(x)
```

```
# 省略代码
return model
```

注意, VGG 中有使用多层同样卷积核数量的卷积层, 因为实验发现多次同样的卷积运算有助于提升平面特征的提取, 这些技巧知识都是实验选拔的结果, 因此不需要过度理解。

## 2. Caffe 实现 VGG-16 模型

由于 `bvlc_googlenet` 没有 VGG-16 模型, 所以本书随书示例代码中录入了一份用于 ImageNet (分类 1000 个类别) 的 VGG-16 模型, 你可以适量修改模型配置, 将它用到你自己的数据训练中。

## 6.5.4 其他模型

目前 Keras 的 applications 支持的模型比较少, 可以看一下:

```
import keras.applications
dir(keras.applications)
```

输出:

```
['InceptionV3',
 'ResNet50',
 'VGG16',
 'VGG19',
 'Xception',
 '__builtins__',
 '__doc__',
 '__file__',
 '__name__',
 '__package__',
 '__path__',
 'imagenet_utils',
 'inception_v3',
 'resnet50',
 'vgg16',
 'vgg19',
 'xception']
```

Caffe 的模型集中在 `caffe/models` 中, 有 Alexnet、Googlenet 等, `caffe/example` 中也有一些简单的模型。

使用 Caffe 的优势在于模块分隔比较好。你完全可以在网上找一些其他高手定制的模型 `prototxt` 文件, 视情况修改一些训练文件 `solver.prototxt` 的参数配置, 就可以将新模型用到你的任务中了, 而不用更改工程的其他部分。

## 6.5.5 回顾

本节介绍了几个 CNN 的实现模型。6.6 节微训练将完成一个通过 Keras 使用上面模型的例子，关于通过 Caffe 使用 CNN 模型的完整例子，请移步第 7 章。

- ImageNet 介绍。
- Inception 结构的模型和 VGG 模型。

## 6.6 微训练模型 (Fine-Tuning)

如果说我看得远，那是因为我站在巨人们的肩上。

——牛顿

6.5 节主要介绍了两个基于 CNN 实现的复杂模型。实际使用中，调试训练一个几千数量级别的图片集的 CNN 模型要花非常多的时间，代价高昂。同时，对于很多任务，数据集往往非常有限，数量上无法满足深度学习模型的拟合要求，有没有办法解决这些问题呢？

本节，我们将使用 Keras 介绍模型的微训练 (Fine-Tuning)，解决上面的问题。

### 备注

本节实现思路借鉴了 Francois Chollet 的博客：<https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html>。

### 6.6.1 二次训练一个成熟的模型

想象这样一件事情，对于一个你从未见过的新事物，你需要花多长时间学习认识它？

图 6-44 是维多利亚时代的“飞猫”，给你几秒钟，你已经可以轻松地理解这一事物的特征：猫脸、尾巴、翅膀。但如果让一个刚出生的孩子学会认识这一事物，需要多长时间呢？

本章的开头提过，刚出生的孩子要经过数年的时间学习认识事物，因为他的脑模型是空白的，并不理解任何类型的特征；而你的脑模型已经被真实世界训练了很多年，经过百亿甚至更大数量级的真实世界的照片样本训练过，能够轻松地理解并识别出很多种事物对象及平面特征，所以接触新事物时，你能很快从中解剖出“熟悉”的特征，掌握识别关键，如图 6-45 所示。



图 6-44 维多利亚时代的“飞猫”



图 6-45 识别多种事物

备注：出自 TED，Fei-Fei Li: How we're teaching computers to understand pictures。

在一个已有能力识别多种特征的成熟模型上，学习辨识新事物将变得很容易。这就是微训练模型（fine-tuning）的原因，二次训练一个已经被深度训练过的模型，在很少的样本和很短的时间情况下，能达到不错的成绩。

## 6.6.2 微训练在 ImageNet 训练好的模型

之前介绍了 ImageNet，这个平台催生了很多优秀的成熟模型。这些模型的参数都在海量的图片样本和庞大的计算机集群中深度训练，可以说是用数值保存的智慧。下面将微训练 ImageNet 数据集上 VGG 模型，达成猫狗识别任务目标。

首先，我们了解一下 ImageNet 训练出来的模型具备识别哪些对象的能力。ImageNet 训练的类别标签有 1000 种，下面列举几种。

ImageNet 训练标签，列举几种如下：



```
{
  0: 'tench, Tinca tinca',
  1: 'goldfish, Carassius auratus',
  2: 'great white shark, white shark, man-eater, man-eating shark,
Carcharodon carcharias',
  3: 'tiger shark, Galeocerdo cuvieri',
  4: 'hammerhead, hammerhead shark',
  5: 'electric ray, crampfish, numbfish, torpedo',
  6: 'stingray',
  7: 'cock',
  8: 'hen',
  9: 'ostrich, Struthio camelus',
  10: 'brambling, Fringilla montifringilla',
  # ...
  990: 'buckeye, horse chestnut, conker',
  991: 'coral fungus',
  992: 'agaric',
  993: 'gyromitra',
  994: 'stinkhorn, carrion fungus',
  995: 'earthstar',
  996: 'hen-of-the-woods, hen of the woods, Polyporus frondosus, Grifola
frondosa',
  997: 'bolete',
  998: 'ear, spike, capitulum',
  999: 'toilet tissue, toilet paper, bathroom tissue'
}
```

完整的标签请访问 <https://gist.github.com/maraoz/388eddec39d60c6d52d4>。

搜索关键词“dog”、“cat”会发现，ImageNet 包含了数个品种的猫狗类别识别，也就是说，ImageNet 训练出来的模型已经具备识别数种猫狗的能力，这对于这个任务而言已经足够了。想象一个成年人，即使他只见过少数几种猫狗，但当他看到一只新品种的猫（或狗）时依然能够知道这是一只猫（或狗），因为不同品种的猫狗的图形特征非常相似。同时，我们的猫狗分类任务对模型的要求很低，只要求识别出猫和狗的品种区别就可以了。

Keras 中，可以用下面的方式加载 VGG 模型在 ImageNet 的训练结果：

```
from keras.applications.inception_v3 import InceptionV3
base_model = InceptionV3(weights='imagenet')
```

补充 applications 的 Model 的几个参数说明：

- include\_top 设置是否包含模型的识别部分，这在本节后面有用。
- weights 可以选择 imagenet 或者 None，对应 ImageNet 训练好的模型和空白模型。第一次加载 ImageNet 训练过的模型时需要联网下载模型参数。



- `input_shape` 模型维度设置，注意在 Keras 中除了第一个输入层外，其余可以空着让模型自行计算维度尺寸。
- `input_tensor` 根据 `input_shape` 设置模型输入，空着让 Keras 自动处理就行。

接下来如何将 ImageNet 训练的模型能力融合到新的模型中，转换成辨别猫狗的能力呢？

## 1. 更换 DNN 模型结构

ImageNet 的任务目标是识别 1000 种事物类别，而我们只需要识别 2 种事物的差异——识别模块的需求不同，所以接下来我们将提取 VGG-16 的卷积层，去掉 ImageNet 的 DNN 模型层，换成适合猫狗辨别任务的 DNN 模型结构，同时冻结 VGG-16 的卷积层参数，开始训练，如图 6-46 所示。

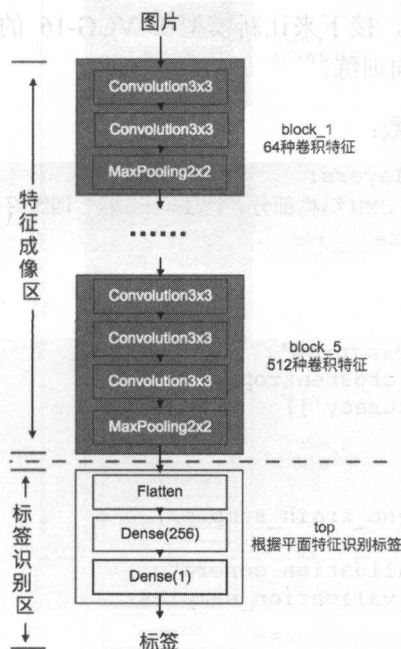


图 6-46 去掉 VGG-16 的 DNN 模块

具体代码如下，首先定义图片输入尺寸：

```
from keras.models import Model
from keras.optimizers import SGD
from keras.applications.vgg16 import VGG16

# 图片尺寸
img_width, img_height = 128, 128
```

```
input_shape = (img_width, img_height, 3)
```

加载 VGG-16 在 ImageNet 的训练权重，不包括 DNN 层：

```
base_model = VGG16(weights='imagenet', include_top=False,
input_shape=input_shape)
```

如图 6-46 所示，在 VGG-16 卷积层输出之后，接入定义好的识别 DNN 层：

```
from keras.layers import Dropout, Flatten, Dense
```

```
x = base_model.output
x = Flatten()(x)
x = Dense(256, activation='relu')(x)
x = Dropout(0.5)(x)
y = Dense(1, activation='sigmoid')(x)
```

```
model = Model(input=base_model.input, output=y)
```

上面组装出了新的模型，接下来让新模型在 VGG-16 的平面特征识别能力基础上，朝“辨别猫狗”这一目标方向训练。

冻结 VGG-16 的训练参数：

```
for layer in base_model.layers:
    # 冻结 VGG 中 ImageNet 的 CNN 结构部分，让 ImageNet 训练好的参数不变
    layer.trainable = False
```

编译、训练模型：

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.fit_generator(
    train_generator,
    samples_per_epoch=nb_train_samples,
    nb_epoch=nb_epoch,
    validation_data=validation_generator,
    nb_val_samples=nb_validation_samples)
```

上面的 DNN 神经元数量并不大，所以速度非常快。50 轮后，成绩在 90% 上下。在一个被海量数据和高效集群训练完毕的成熟模型上，新模型轻松突破了原来的瓶颈。不错的提升！

注意，随时保存训练结果是个好习惯，我们用 `save_weights` 和 `load_weights` 实现保存和恢复：

```
model.save_weights('merge_model.h5')
```

## 2. 微调连接部分

毕竟 ImageNet 的样本集合任务目标和这里的任务目标是有差异的，因此像我们这样直接粗暴地将 ImageNet 的 CNN 模块和猫狗辨别的 DNN 模块放一起训练，会不会有什么问题呢？这就像一个人，为了提高步行速度直接嫁接了长跑运动员的大腿，一定能适应吗？

解决的方法很简单，只需要动一点小手术，微调两者的结合部分即可。具体的实现是，准备好 ImageNet 训练好的 CNN 模块和上面训练好的 DNN 模块，连接起来组成 VGG-16；接着冻结 ImageNet 部分的前几层 CNN 模块，让猫狗辨别任务的 DNN 模块和 ImageNet 的 CNN 模块的衔接部分一起在比较低的速率下再训练一段时间（低梯度学习速率，使梯度下降中每一次 epoch 参数变化的幅度很低），如图 6-47 所示。

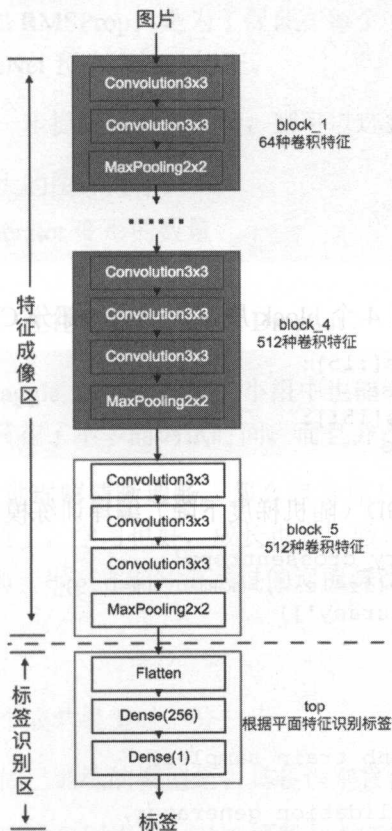


图 6-47 微调训练连接部分模块

具体实现，先看一下模型现在具体在多少层：

```
for i, layer in enumerate(model.layers):
    print(i, layer.name)
```

输出:

```
(0, 'input_1')
(1, 'block1_conv1')
(2, 'block1_conv2')
(3, 'block1_pool')
(4, 'block2_conv1')
(5, 'block2_conv2')
(6, 'block2_pool')
(7, 'block3_conv1')
(8, 'block3_conv2')
(9, 'block3_conv3')
(10, 'block3_pool')
(11, 'block4_conv1')
(12, 'block4_conv2')
(13, 'block4_conv3')
(14, 'block4_pool')
(15, 'block5_conv1')
(16, 'block5_conv2')
(17, 'block5_conv3')
(18, 'block5_pool')
(19, 'flatten_2')
(20, 'dense_2')
(21, 'dropout_1')
(22, 'dense_3')
```

如图 6-47 所示, 冻结前 4 个 block 层, 让最后一部分 CNN 层和 DNN 一起微调训练:

```
for layer in model.layers[:15]:
    layer.trainable = False
for layer in model.layers[15:]:
    layer.trainable = True
```

注意, 要使用低速的 SGD (随机梯度下降) 编译训练模型:

```
model.compile(loss='binary_crossentropy',
              optimizer=SGD(lr=1e-4, momentum=0.9),
              metrics=['accuracy'])

# 微调训练
model.fit_generator(
    train_generator,
    samples_per_epoch=nb_train_samples,
    nb_epoch=nb_epoch,
    validation_data=validation_generator,
    nb_val_samples=nb_validation_samples)
```

50 轮后, 训练的成绩有 3%~4% 的提升, 还不错。虽然工程实现很简单, 但这些实现的背后包含了一些深刻且有意义的思考。

(1) 上面用训练好的 DNN 层和 VGG-16 的卷积层对接微调, 是因为如果用一个空白

的 DNN 层直接和 VGG-16 的卷积层对接训练, 则梯度下降时随机化的 DNN 层参数通过 BP 反馈会将已训练好的 VGG 16 直接打乱到接近无序, 这样之前加载预先训练的 VGG-16 的模型就完全失去了意义。

(2) 上面我们选择只微调最后的卷积块, 而不是整个网络, 并且使用很低的速率, 这是为了防止过拟合。对于猫狗辨别任务而言, VGG-16Net 完全大材小用, 整个网络具有过剩的识别能力, 因此很容易过拟合。同时根据层级特征的特性, 前几层模型学习的特征很基础, 对应的类别能力也很模糊。比如特征可能是线条、边缘, 等等, 这些特征距离猫狗对象的识别还很远, 所以不需要变。我们只对最后一层进行调整就可以了。

(3) 微调模型的连接部分应该在很低的学习率下进行, 选择使用 SGD 而不是其他能够变化学习率的优化算法 (如 RMSProp) 是为了保证在每个 epoch 下, 参数的更新幅度不要太大, 以免过度破坏 ImageNet 预先训练的特征。

接下来, 如果还想更进一步提高模型的成绩, 则可以选择以下几个方向尝试:

- 更大的样本集, 更大的图片尺寸。
- 扩大 ImageDataGenerator 变形的数量。
- 更大的 Dropout。
- 扩大微调的层范围, 往往需要同时增大 Dropout。

至此, 模型的成绩在 Kaggle 的各个国家的小组中也能有不错的排名。前几名的成绩在 98% 上下, 但想想, 我们只花了不多的调试时间, 而且并没有用全部的样本数据。

如果读到这里你依然感觉理解清晰通畅, 那么恭喜你! 在工程应用领域, 你已经成为一个优秀模型专家的潜质了。学习也好, 成长也好, 做许多事情都好比小学课文中的“小马过河”, 只有当你走到那一步, 才能知道趟过的河流深浅。

### 6.6.3 回顾

总结一下, 在微训练一个成熟模型时, 分三步:

- (1) 查询标签类别, 评估已训练的模型是否具备识别目标任务特征的能力。
- (2) 如果具备, 更换模型的识别模块, 对接模型, 训练模型识别模块的参数。
- (3) 微调连接部分。



# Caffe 实例：狗狗品种辨别

本章，我们将从数据爬取到识别，实现一个完整的 Caffe 图片识别项目。

任务目标是训练一个能够识别图片中狗狗品种模型，7.3 节将运用训练好的模型识别狗狗的生活照。

## 说明

本章完整项目地址：<https://github.com/bbfamily/DogJudge>。

机器学习环境部署见附录 A，深度学习环境部署见附录 B，随书示例代码运行环境部署见附录 C。

## 7.1 准备图片数据

首先我们从百度图片搜集狗狗的图片，准备数据集。

### 7.1.1 搜集狗狗图片

在本书的随书代码中，使用 Selenium 配合 BeautifulSoup、requests 爬取图片，达到目标数量或者得到所有图片停止。对这些技术感兴趣的读者可以查阅 SpiderBdImg 源码。

```
SpiderBdImg.spider_bd_img(  
    [u'拉布拉多', u'哈士奇', u'金毛', u'萨摩耶', u'柯基', u'柴犬',  
     u'边境牧羊犬', u'比格', u'德国牧羊犬', u'杜宾', u'泰迪犬',  
     u'博美', u'巴哥', u'牛头梗'],  
    use_cache=True)
```

搜集的狗狗图片按照类别存放在不同的目录中，如下所示：

```
makedirs ../gen/baidu/image/金毛  
makedirs ../gen/baidu/image/哈士奇
```

```
makedirs ../gen/baidu/image/拉布拉多
```

```
.....
```

```
makedirs ../gen/baidu/image/牛头梗
```

## 7.1.2 清洗数据

搜索引擎的返回结果不一定百分百和检索词相关，所以爬取图片后，我们需要人工扫描一下图片，把不相关的删掉。如图 7-1 所示。



图 7-1 删除下载错误的图片

也有一些残了的图片，这些都是坏样本，也需要删掉。如图 7-2 所示。

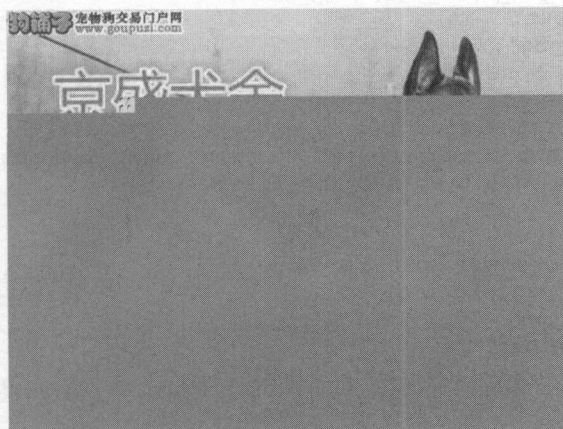


图 7-2 删除残图

如果数据集非常大，那么上面这两步就需要通过代码过滤了。由于这里的样本集合

不大，所以我们就人工完成了。

### 7.1.3 标准化数据

下载的图片格式包括 png、jpg 等很多种，这里需要使用统一格式，即用 ImgStdHelper 代码将所有图片都转成 jpeg 格式。

ImgStdHelper 代码如下：

```
"""
    标准化图片格式
"""
import glob
import imghdr
import os
import PIL.Image
from PIL import ImageFile

__author__ = 'BBFamily'

ImageFile.LOAD_TRUNCATED_IMAGES = True

def std_img_from_root_dir(root_dir, a_ext):
    """
    将 ext 目录下的所有图片全部转换为 jpeg
    """
    img_list = find_img_by_ext(a_ext, root_dir)
    all_ext = change_to_real_typed(img_list)
    for ext in all_ext:
        if ext != 'jpeg':
            if ext is None:
                ext = a_ext
            sub_img_list = find_img_by_ext(ext, root_dir)
            _ = map(lambda img: covert_to_jpeg(img), sub_img_list)
            change_to_real_typed(sub_img_list)

def covert_to_jpeg(org_img, dst_img=None):
    im = PIL.Image.open(org_img)
    if dst_img is None:
        dst_img = org_img
    im.convert('RGB').save(dst_img, 'JPEG')

def find_img_by_ext(ext, root_dir):
    """
    只遍历根目录及一级子目录
    """
```

```

"""
dirs = [root_dir + name for name in os.listdir(root_dir) if
        os.path.isdir(os.path.join(root_dir, name))]
dirs.append(root_dir)
img_list = list()
for dr in dirs:
    sub_list = glob.glob(dr + "/*.") + ext)
    img_list.extend(sub_list)
return img_list

def change_to_real_tye(img_list):
    all_type = []
    for img in img_list:
        real_type = imghdr.what(img)
        if all_type.count(real_type) == 0:
            all_type.append(real_type)
        if real_type is None:
            continue
        real_name = img[:img.rfind('.')] + '.' + real_type
        os.rename(img, real_name)
    return all_type

```

使用 `ImgStdHelper`:

```

import ImgStdHelper

ImgStdHelper.std_img_from_root_dir('../gen/baidu/image/', 'jpg')

```

运行成功后所有图片均为 `jpeg` 格式。至此，待训练的图片样本全部准备完毕。

## 7.1.4 回顾

本节介绍了图片数据的准备工作。

- 爬取数据
- 清洗数据
- 标准化图片格式

## 7.2 训练模型

7.1 节搜集了狗狗的图片样本，下载的图片样本按狗狗类别文件夹分别存放，本节将使用 Caffe，完成 `GoogleLenet` 模型的训练。



## 7.2.1 生成样本集

DogType.sh 截取图片路径中的狗狗类别名称，生成样本集描述文件 data.txt。

data.txt 文件：

```
DATA=../gen/baidu/image

HASHIQI=${DATA}/哈士奇
LALA=${DATA}/拉布拉多
BM=${DATA}/博美
CQ=${DATA}/柴犬
DM=${DATA}/德国牧羊犬
DB=${DATA}/杜宾

OUTPUT=../gen/dog_judge
mkdir $OUTPUT

echo "Create data.txt..."
rm -rf $OUTPUT/data.txt

find $HASHIQI -name *.jpeg | cut -d '/' -f 5,6 | sed "s/$/ 1/">>$OUTPUT/data.txt
find $LALA -name *.jpeg | cut -d '/' -f 5,6 | sed "s/$/ 2/">>$OUTPUT/lala.txt
find $BM -name *.jpeg | cut -d '/' -f 5,6 | sed "s/$/ 3/">>$OUTPUT/bm.txt
find $CQ -name *.jpeg | cut -d '/' -f 5,6 | sed "s/$/ 4/">>$OUTPUT/cq.txt
find $DM -name *.jpeg | cut -d '/' -f 5,6 | sed "s/$/ 5/">>$OUTPUT/dm.txt
find $DB -name *.jpeg | cut -d '/' -f 5,6 | sed "s/$/ 6/">>$OUTPUT/db.txt

cat $OUTPUT/lala.txt>>$OUTPUT/data.txt
cat $OUTPUT/bm.txt>>$OUTPUT/data.txt
cat $OUTPUT/cq.txt>>$OUTPUT/data.txt
cat $OUTPUT/dm.txt>>$OUTPUT/data.txt
cat $OUTPUT/db.txt>>$OUTPUT/data.txt

rm -rf $OUTPUT/lala.txt
rm -rf $OUTPUT/bm.txt
rm -rf $OUTPUT/cq.txt
rm -rf $OUTPUT/dm.txt
rm -rf $OUTPUT/db.txt
```

运行 DogType.sh 脚本，生成的数据如下：

```
data_path = '../gen/dog_judge/data.txt'
print open(data_path).read(100)
```

输出：



```

哈士奇/001e5dd0f5aa0959503324336f24a5ea.jpeg 1
哈士奇/001eae03d6f282d1e9f4cb52331d3e20.jpeg 1
.....

```

接着，生成数字类别对应的 label 文件：

```

import pandas as pd
import numpy as np

class_map = pd.DataFrame(np.array([[1, 2, 3, 4, 5, 6],
                                     ['哈士奇', '拉布拉多', '博美', '柴犬',
                                      '德国牧羊犬', '杜宾']]).T,
                          columns=['class', 'name'],
                          index=np.arange(0, 6))
class_map.to_csv('../gen/class_map.csv', columns=class_map.columns,
                  index=True)

```

## 7.2.2 生成训练、测试数据集

下面 TrainValSplit 将待训练的数据集每个类别按照 `n_folds=10`，即分成 10 份，val 占 1 份，train 占 9 份，TrainValSplit 用法和 scikit-learn 中分割参数 `n_folds` 的用法一样。

在 `gen` 下重新生成训练数据集、测试数据集，交织测试数据集，这里的 `test` 与 `val` 数据一样，不过 `test` 没有分类标注。

TrainValSplit 文件内容如下：

```

def train_val_split(data_path, n_folds=10):
    if n_folds <= 1:
        raise ValueError('n_folds must > 1')

    with open(data_path, 'r') as f:
        lines = f.readlines()
        class_dict = defaultdict(list)
        for line in lines:
            cs = line[line.rfind(' '):]
            class_dict[cs].append(line)

    train = list()
    val = list()
    for cs in class_dict:
        cs_len = len(class_dict[cs])
        val_cnt = int(cs_len / n_folds)
        val.append(class_dict[cs][:val_cnt])
        train.append(class_dict[cs][val_cnt:])
    val = list(itertools.chain.from_iterable(val))
    train = list(itertools.chain.from_iterable(train))
    test = [t.split(' ')[0] for t in val]

```

```

fn = os.path.dirname(data_path) + '/train_split.txt'
with open(fn, 'w') as f:
    f.writelines(train)
fn = os.path.dirname(data_path) + '/val_split.txt'
with open(fn, 'w') as f:
    f.writelines(val)
fn = os.path.dirname(data_path) + '/test_split.txt'
with open(fn, 'w') as f:
    f.writelines(test)

```

### TrainValSplit 使用:

```
import TrainValSplit
```

```
TrainValSplit.train_val_split(data_path, n_folds=10)
```

### 看一下分割的结果:

```

train_path = '../gen/dog_judge/train_split.txt'
with open(train_path) as f:
    print 'train set len = {}'.format(len(f.readlines()))
val_path = '../gen/dog_judge/val_split.txt'
with open(val_path) as f:
    print 'val set len = {}'.format(len(f.readlines()))

```

### 输出:

```

train set len = 9628
val set len = 1066

```

## 7.2.3 生成 lmdb

这里的 caffe 安装路径是 “/root/caffe”。

DogLmdb.sh 将样本集转成 lmdb 数据库:

```

echo "Begin lmdb..."
ROOTFOLDER=../gen/baidu/image/
OUTPUT=../gen/dog_judge
CONVERT_BIN=/root/caffe/build/tools/convert_imageset

rm -rf $OUTPUT/img_train_lmdb
$CONVERT_BIN --shuffle \
--resize_height=256 --resize_width=256 \
$ROOTFOLDER $OUTPUT/train_split.txt $OUTPUT/img_train_lmdb

rm -rf $OUTPUT/img_val_lmdb
$CONVERT_BIN --shuffle \
--resize_height=256 --resize_width=256 \
$ROOTFOLDER $OUTPUT/val_split.txt $OUTPUT/img_val_lmdb
echo "Done lmdb..."

```

在 Caffe 自带的 Googlenet 的模型配置文件中，输入图片样本的 `crop_size` 大小为 224，所以这里将输入图片尺寸修改到  $256 \times 256$ （大于等于 224），让样本适配模型格式。如果设置了不同的输入尺寸，则需要从上到下重新计算生成各层的输入输出格式。

运行 `DogLmdb.sh`，发现处理一些样本显示为“Could not open or find file”，这些是 7.1 节没有清洗掉的坏样本，不需要在意。

## 7.2.4 生成去均值文件

`DogMean.sh` 生成去均值文件，去均值可以加速模型训练。

`DogMean.sh` 文件内容：

```
echo "Begin mean..."

LMDB=../gen/dog_judge/img_train_lmdb
MEAN_BIN=/root/caffe/build/tools/compute_image_mean
OUTPUT=../gen/dog_judge/mean.binaryproto

echo $OUTPUT

$MEAN_BIN $LMDB $OUTPUT

LMDB=../gen/dog_judge/img_val_lmdb
OUTPUT=../gen/dog_judge/mean_val.binaryproto
echo $OUTPUT
$MEAN_BIN $LMDB $OUTPUT
```

补充说明：在本地环境中，需要注意替换脚本的路径。

运行 `DogMean.sh`，生成 `mean.binaryproto` 和 `mean_val.binaryproto` 两个文件。

## 7.2.5 更改 prototxt 文件

7.2.4 节生成的图片尺寸是  $256 \times 256$ ，这里不更改 Googlenet 的模型结构，直接复用“/你的 Caffe 目录/models/bvlc\_googlenet”下的 `prototxt` 配置文件，其中与训练相关的两个 `prototxt` 文件是 `train_val.prototxt` 和 `solver.prototxt`。`train_val.prototxt` 文件描述模型（训练—测试）网络结构，`solver.prototxt` 文件描述训练参数。

### 1. 更改 train\_val.prototxt

要改的几个地方如下。

(1) 在 TRAIN 和 TEST 层的参数描述中更改 `mean_file` 路径，换成上面生成的文件路径：

```
transform_param {
  mirror: true
  crop_size: 224
  mean_file: "/root/maxmon/DogJudge/gen/dog_judge/mean.binaryproto"
} data_param {
  source: "/root/maxmon/DogJudge/gen/dog_judge/img_train_lmdb"
  batch_size: 32
  backend: LMDB
}
```

可以考虑把 TRAIN 或者 TEST 的 `mirror: false` 改成 `true`，因为样本集不大。

训练集样本数大约为  $1000=50 \times 20$ ，所以这里的 `batch_size=50`，对应 `solver.prototxt` 的 `test_iter` 改为 20。

(2) 更改输出维度，原模型文件是对应 ImageNet 的 1000 个类别的识别，搜索 “`num_output: 1000`”，将几个 “`loss/classifier`” 层的输出维度参数改为 6：

```
inner_product_param {
  num_output: 6 # 6 个类别
  weight_filler {
    type: "xavier"
  }
  bias_filler {
    type: "constant"
    value: 0
  }
}
```

由于这里只改最后的输出维度，所以不会影响其他层参数。如果更改输入图片尺寸，则需要重新生成对应的模型维度参数。

## 2. 更改 solver.prototxt

(1) `test_iter: 20` 测试的时候，一次迭代 50 个样本，迭代 20 次。

(2) 训练集约 10000，所以 `test_interval: 100`。即训练  $100 \times 50$  (`batch_size`) 个样本时，测试一次。

(3) `display: 100`，每 100 迭代显示一次当前成绩。

(4) `snapshot: 5000`，每 5000 次迭代保存训练结果。

(5) `snapshot_prefix`，保存训练结果的路径。

(6) `solver_mode: CPU`，视情况更改 CPU 或者 GPU。



## 7.2.6 训练模型

接下来，可以使用 DogTrain.sh 训练模型，训练日志很多，所以这里不再展示。DogTrain.sh 脚本如下：

```
echo "Begin train...."

CAFEBIN=/root/caffe/build/tools/caffe
SOLVER=../pb/solver.prototxt
$CAFEBIN train -solver $SOLVER

echo "Done train..."
```

这样，就完成了模型的训练。7.3 节将展示训练的结果。

## 7.2.7 回顾

本节介绍了使用 Caffe 训练模型的过程。

- 生成 train、val 数据集。
- 生成 lmdb 数据库。
- 去均值。
- 更改 train\_val.prototxt、solver.prototxt 文件。
- 训练模型。

## 7.3 使用生成的模型进行分类

本节将测试 7.2 节训练好的模型，我们使用 22 张拉布拉多犬的真实生活照片测试模型成绩。

### 7.3.1 更改 deploy.prototxt

首先，更改“/你的 Caffe 目录/models/bvlc\_googlenet”下的 deploy.prototxt 文件，将“loss3/classifier”的“num\_output: 1000”替换成“num\_output: 6”：

```
inner_product_param {
  num_output: 6 # 输出类别数
  weight_filler {
    type: "xavier"
  }
  bias_filler {
    type: "constant"
```



```

        value: 0
    }
}

```

### 7.3.2 加载模型

接下来，加载训练结果，生成模型：

```

import caffe

caffe.set_mode_cpu()

# 模型文件
model_def = '../pb/deploy.prototxt'
# 训练结果
model_weights = '../gen/dog_judge/dog_judge_train_iter_5000.caffemodel'
# 均值文件
model_mean_file = '../gen/dog_judge/mean.binaryproto'

net = caffe.Net(model_def, model_weights, caffe.TEST)

# 均值处理
mean_blob = caffe.proto.caffe_pb2.BlobProto()
mean_blob.ParseFromString(open(model_mean_file, 'rb').read())
mean_numpy = caffe.io.blobproto_to_array(mean_blob)
mu = mean_numpy.mean(2).mean(2)[0]
print 'mu = {}'.format(mu)

# 颜色格式处理
transformer = caffe.io.Transformer({
    'data': net.blobs['data'].data.shape})
transformer.set_transpose('data', (2, 0, 1))
transformer.set_mean('data', mu)
transformer.set_raw_scale('data', 255)
transformer.set_channel_swap('data', (2, 1, 0))
mu = [116.2626216  129.17550814 137.46700908]

```

可以通过下面的代码看一下模型目前各层的输入输出格式：

```

for layer_name, blob in net.blobs.iteritems():
    print layer_name + '\t' + str(blob.data.shape)

```

由于输出内容过长，所以这里不再展示，读者可查阅随书示例代码运行结果。

主角终于要上场了，使用拉布拉多犬的生活照作为测试样本，看看准确率怎么样。

看看狗狗类别：

```

class_map = pd.read_csv('../gen/class_map.csv', index_col=0)
class_map

```

输出如表 7-1 所示。

表 7-1 狗狗类别

	class	name
0	1	哈士奇
1	2	拉布拉多
2	3	博美
3	4	柴犬
4	5	德国牧羊犬
5	6	杜宾

22 张生活照：

```
predict_dir = '../abu'
img_list = glob.glob(predict_dir + '/*.jpeg')
len(img_list)
```

输出：

22

开始预测狗照：

```
error_prob = []
for img in img_list:
    image = caffe.io.load_image(img)
    transformed_image = transformer.preprocess('data', image)
    plt.imshow(image)
    plt.show()
    net.blobs['data'].data[...] = transformed_image
    output = net.forward()
    output_prob = output['prob'][0]
    print 'predicted class is:', \
        class_map[class_map['class'] == \
            output_prob.argmax()].name.values[0]
    if output_prob.argmax() != 2:
        error_prob.append(img)
```

输出如下（下面仅展示部分，完整见随书示例代码），如图 7-3 至图 7-17 所示。

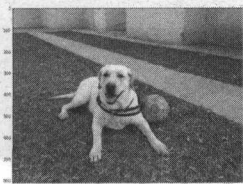


图 7-3 狗照 1

predicted class is: 拉布拉多

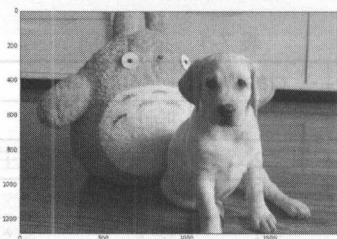


图 7-4 狗照 2

predicted class is: 拉布拉多

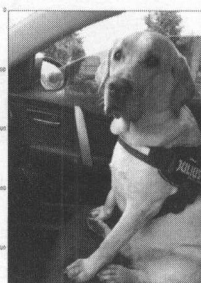


图 7-5 狗照 3

predicted class is: 拉布拉多



图 7-6 狗照 4

predicted class is: 拉布拉多



图 7-7 狗照 5

predicted class is: 德国牧羊犬

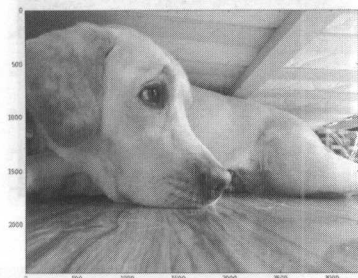


图 7-8 狗照 6

predicted class is: 博美

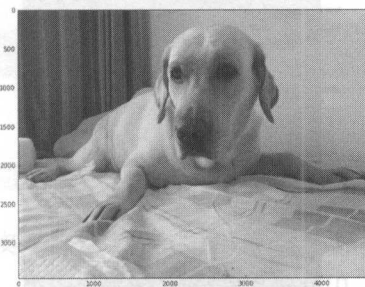


图 7-9 狗照 7

predicted class is: 拉布拉多



图 7-10 狗照 8

predicted class is: 拉布拉多

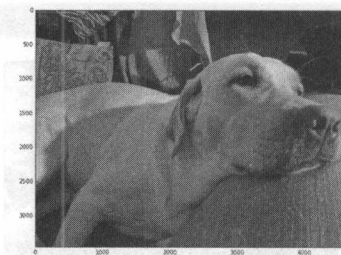


图 7-11 狗照 9

predicted class is: 拉布拉多

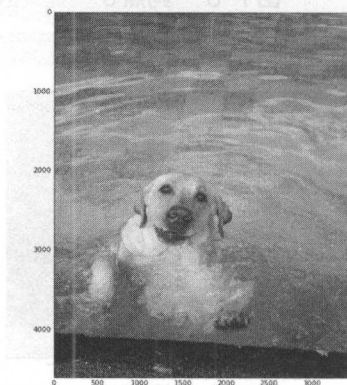


图 7-12 狗照 10

predicted class is: 拉布拉多

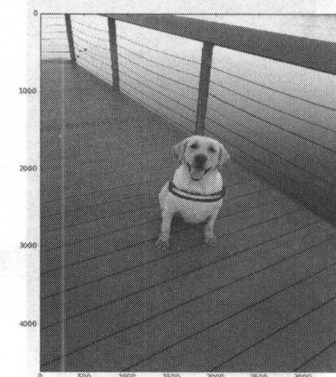


图 7-13 狗照 11

predicted class is: 杜宾



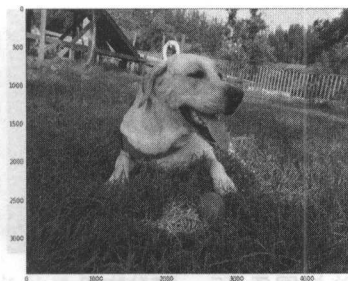


图 7-14 狗照 12

```
predicted class is: 拉布拉多
```

看下准确率:

```
accuary = (len(img_list) - len(error_prob))/float(len(img_list))
accuary
```

输出:

```
0.8181818181818182
```

能达到 80% 的准确率其实是出乎预料的。在数据不算多，且质量一般的情况下能达到这种效果，Caffe 训练不错。有些照片，比如，图 7-10 中躺着睡觉耳朵立起来的那个都判断对了，即使人去看也有可能认成是哈士奇。另外，本示例特意避开了金毛，因为笔者也经常分不清楚金毛和拉布拉多……

再看一遍分错的这几个，感觉错的 rank 基本符合正态分布，没什么可特别挖掘的。

```
for img in error_prob:
    try:
        image = caffe.io.load_image(img)
    except Exception:
        continue
    transformed_image = transformer.preprocess('data', image)
    plt.imshow(image)
    plt.show()
    net.blobs['data'].data[...] = transformed_image
    output = net.forward()
    output_prob = output['prob'][0]
    top_inds = output_prob.argsort()[::-1]
    for rank, ind in enumerate(top_inds, 1):
        print 'probabilities rank {} label is {}'.format(rank,
            class_map[class_map['class'] == ind].name.values[0])
```

输出:



图 7-15 错分狗照 1

```

probabilities rank 1 label is 德国牧羊犬
probabilities rank 2 label is 杜宾
probabilities rank 3 label is 拉布拉多
probabilities rank 4 label is 柴犬
probabilities rank 5 label is 博美
probabilities rank 6 label is 哈士奇

```



图 7-16 错分狗照 2

```

probabilities rank 1 label is 博美
probabilities rank 2 label is 柴犬
probabilities rank 3 label is 拉布拉多
probabilities rank 4 label is 哈士奇
probabilities rank 5 label is 杜宾
probabilities rank 6 label is 德国牧羊犬

```

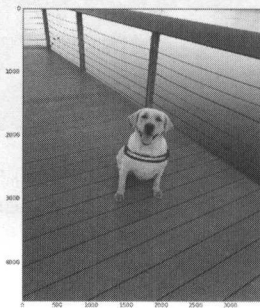


图 7-17 错分狗照 3

```
probabilities rank 1 label is 杜宾  
probabilities rank 2 label is 拉布拉多  
probabilities rank 3 label is 德国牧羊犬  
probabilities rank 4 label is 柴犬  
probabilities rank 5 label is 博美  
probabilities rank 6 label is 哈士奇
```

OK，就到这里。我们准备的数据集很小，如果你希望更进一步提高模型的识别能力，可以考虑以下几个方面：

- 爬取更多图片。
- 使用 OpenCV 或者 Keras 的 API 生成变形的图片样本，扩大训练集。
- 花更多的时间微调模型 Dropout 参数等。
- 尝试微调训练 ImageNet 模型。

针对小数据集，运用数据技巧，合理训练、调试，深度学习模型也能做得不错。

### 7.3.3 回顾

本节介绍了使用 Caffe 加载已训练好的模型，分类新样本的过程。

- 更改 deploy.prototxt 文件。
- 加载生成分类模型。

# 漫谈时间序列模型

我们通过视觉、听觉、触摸、味道、气味感知这个鲜活的世界，相比人类的五感，机器感知外界信息的方式，主要划分为三种：向量、空间数据和时间序列。其中，向量数据又可以视为一维的空间数据或者一维的时间序列。如图 8-17 所示。

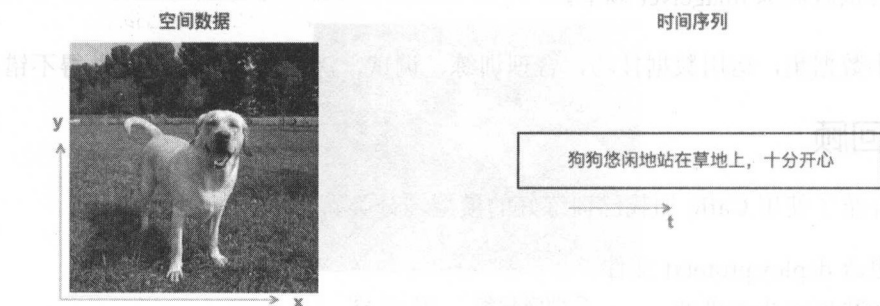


图 8-1 空间数据和时间序列

数据在时间上表现为随着时间流动的形式，也就是序列。和空间聚集形式的数据比较，序列的最大特性不再是局部的聚集（虽然也有局部聚集的特性），而是对序列前文的记忆——序列中每个单元需要结合前文，才能形成事物特征。

如图 8-1 所示，“狗狗悠闲地站在草丛中，十分开心”。从左往右读，你读到的每一个单词都需要联系前面的单词才能表达出对应的意思。这就是序列的时序特性，类似于人脑在阅读时对上下文语境的短期“记忆”。

第 6 章我们介绍了 CNN 层，借助卷积运算，以空间的方式关联待训练的参数，提取空间特征。本章，我们将介绍从时间上关联参数的模型——RNN。

本章上场两个主角。

- Embedding：生成时间序列的特征向量。
- RNN：有效处理特征向量序列的神经元模型。

在工程应用范围上，时间模型和空间模型涉及的技术知识其实完全不是一个量级。CNN 目前主要落地在图片中，是机器学习技术的缩小范围应用。而时间序列相关的模型技术可以落地到视频、语音、NLP、甚至金融等多个领域，是机器学习技术对外的扩散应用。本章篇幅有限，只挑其中一些点作一个漫谈，抛砖引玉，有兴趣的读者可以继续自行探索。

## 8.1 Embedding

从根本上讲，文字是表示声音的形状。因此，它们代表着通过眼睛这扇窗口而进入观念事物。

——《信息简史》

本节介绍 Embedding (Embedding 中文直译为“嵌入”，让人感觉很怪，所以我们用英文称呼它)。

数据可以承载的意义多种多样，如声音、文本、单词、多媒体视频流、金融数据流，等等，这些数据可以被编码成序列的形式，比如按单词集 One-Hot 编码的文本向量序列、采样音频序列的傅里叶变换、图片像素数组序列等。

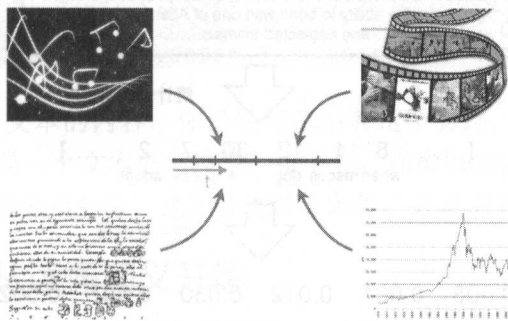


图 8-2 时间序列数据

Embedding 泛指一种转换方式，将数据编码的序列转换成更合理的特征向量或者特征向量序列。比如处理单词序列的 word2vec 将 One-Hot 编码的单词转换成词义特征向量，视频流的 Embedding 则通过对每个图层应用卷积生成特征向量序列。从这个意义上理解，CNN 可以说是图片像素序列的 Embedding 实现。

本节将简单介绍文本、单词的 Embedding 实现。



### 8.1.1 简单的文本识别

首先，如何帮助机器读文章？

先介绍一种简单的人工特征的做法。在机器的视野中，所有的样本事物都是通过向量描述的，所以我们的目标就是把一段短文本转换成机器能够读懂的数值向量。

看一篇文章：

Without a doubt, a loving and friendly puppy or dog can put an instant smile on your face! When you adopt a dog from Atlanta Humane Society, you gain a wonderful canine companion. But most of all, when you adopt a rescue dog, you have the ability to bond with one of Atlanta's forgotten and neglected animals..... (省略)

这是亚特兰大人道协会关于“ADOPT YOUR NEW BEST FRIEND—A DOG OR PUPPY (领养你最好的朋友——宠物犬或者小狗)”网文的摘录。我们希望把这些文本信息转换成向量，让机器能“读懂”文章。一种常见的做法就是按单词出现的次数统计向量。如图 8-3 所示。

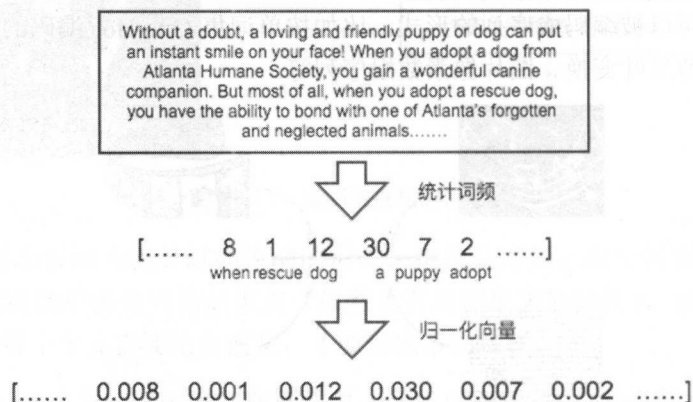


图 8-3 词频统计向量

代码实现：

```
import re
import numpy as np
```

```
def process_text(text):
    """将标点符号替换成空格"""
    dot_word = r'。|，|\.|!|?|,|!|,|\(|\)|\(|\)|'
    return re.sub(dot_word, ' ', text)
```

```
def text2vec(text):
    """将文本转换成向量"""
    cleaned_text = process_text(text) # 输入文本预处理
    text_vec = cleaned_text.split()
    vocab_list = list(set(text_vec)) # 词汇集
    numer_vec = [0.] * len(vocab_list) # 数字向量
    for word in text_vec:
        # 每遇到一个单词，对应的数字向量位置+1
        numer_vec[vocab_list.index(word)] += 1
    return numer_vec / np.sum(numer_vec) # 归一化，让向量中所有的数值加起来等于1
text = 'Without a doubt, a loving and friendly puppy or dog can' \
    ' put an instant smile on your face! When you adopt a dog ' \
    'from Atlanta Humane Society, you gain a wonderful canine ' \
    'companion. But most of all, when you adopt a rescue dog, ' \
    'you have the ability to bond with one of Atlanta's ' \
    'forgotten and neglected animals.'
text2vec(text)
```

输出：

```
array([0.00321543, 0.00321543, 0.20257235, 0.00643087, 0.00321543,
       0.00321543, 0.00321543, 0.00321543, 0.00643087, 0.08360129,
       0.02250804, 0.0096463, 0.05787781, 0.03858521, 0.02250804,
       0.02250804, 0.04180064, 0.0192926, 0.0192926, 0.03536977,
       0.09003215, 0.08038585, 0.02250804, 0.0192926, 0.02250804,
       0.04180064, 0.07073955, 0.0096463, 0.00643087, 0.02893891])
```

生成的向量描述了文本的内容，你已经可以沿着这个套路，在模型后面接上分类层（比如 DNN 层），这就是一个文本分类模型。接下来，你可以尝试收集文本数据，按上面的思路完成一个文本情景分类任务，比如分类垃圾邮件、分类书籍类别等。

## 8.1.2 深度学习从读懂词义开始

OK，计算机能够简单玩转文本识别了。但是基于“词频统计”这一方式构造出的文本向量就像一个人工设计出的特征一样，是对文本的硬描述，一种浅层模型的玩法。这种玩法无法胜任一些更复杂、更抽象的语言任务，比如文本翻译、看图说话等。

我们希望计算机真正能够在一定程度“读懂”一个文本内在的含义，进而完成一些复杂的语言任务。深度学习先从教计算机理解一个单词的含义开始。对计算机而言，理解词义就是生成一个合理描述单词特征的向量。

1. word2vec 目标

所以我们的目标很简单：生成单词的特征向量，而 Google 的开源深度学习工具：word2vec 完美地解决了这个问题。

假设文章总共 1000 个单词，那么 One-Hot 编码下每个单词向量长度为 1000 维，每个向量只含有一个数值 1。比如上文的 “puppy” 可以是  $\begin{bmatrix} 0 \\ \vdots \\ 1 \end{bmatrix}$ ，One-Hot 编码的向量显然无法描述词义，我们可以将 One-Hot 编码的向量输入到训练好的 word2vec 模型，这样就可以输出一个 “puppy” 的特征向量，如  $\begin{bmatrix} 0.2 \\ \vdots \\ 0.4 \end{bmatrix}$ 。

2. word2vec 原理

简单解释下 word2vec，它的本质就是：编码单词的上下文描述单词的词义。

举个例子，对于单词 “dog”，假设数值化狗的属性是：看家 0.2、摇尾巴 0.1、打滚 0.2、撕咬 0.1、汪汪 0.4，等。那么 word2vec 模型通过训练海量文本，发掘出合理上下文

单词的分配权重，即生成的狗的特征向量就是  $\begin{bmatrix} 0.2 \\ 0.1 \\ 0.1 \\ 0.4 \end{bmatrix}$ 。如图 8-4 所示。

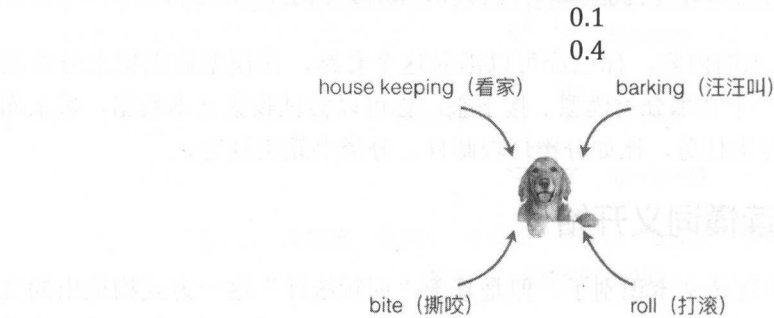


图 8-4 如何定义单词 “狗”

上下文表示的词义向量除了能表达词义本身外，还能很好地描述词与词之间的关系。因为事物与事物之间有相似的属性。单词 “狗” 与 “猫” 如图 8-5 所示。

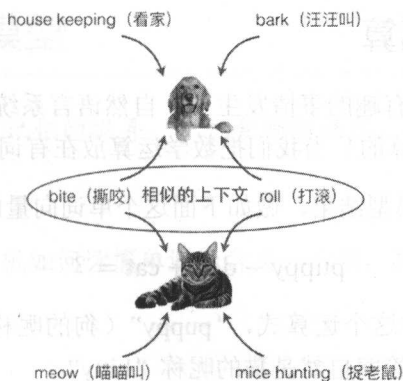


图 8-5 单词“狗”与“猫”

总体来说，word2vec 可以做好两件事：

- 表示单词的词义。
- 表示词与词之间的关系。

### 3. 使用 word2vec

使用 Google 预训练好的 word2vec 模型需要：

- word2vec 工具本身，推荐 Python 接口的 gensim，可以使用 pip 安装，详见官网。
- Google 训练好的模型结果。

(<https://drive.google.com/uc?id=0B7XkCwpI5KDYNINUTTISS21pQmM&export=download>，约 1.5GB 大小)

你可以自己训练 word2vec 模型，前提是准备好足够量的语料数据。word2vec 模型是一个深度学习分类模型，用上下文预测中间单词（也可以反过来）。读者可以自行探索这些知识。CBOW 模型——构造 word2vec 待分类样本如图 8-6 所示。

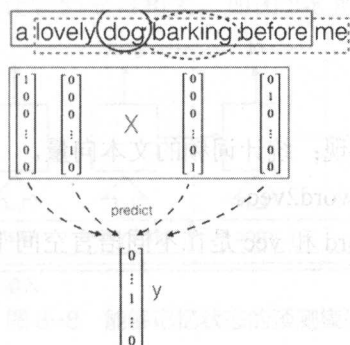


图 8-6 CBOW 模型——构造 word2vec 待分类样本

### 8.1.3 游戏：词义运算

在 word2vec 中，一些有趣的事情发生了。自然语言系统中的单词变为了数值向量，而在数学上，向量是可以运算的！当我们把数学运算放在有词义的向量上会发生什么呢？

加载 Google 训练好的模型结果，做如下面这个单词向量的运算式：

$$\text{puppy} - \text{dog} + \text{cat} = ?$$

如果放在特征层面思考这个运算式，“puppy”（狗的昵称）减去“dog”狗的部分，加上“cat”猫的部分，那么答案自然是猫的昵称“kitty”。

$$\text{puppy} - \text{dog} + \text{cat} = \text{kitty}$$

另一个例子就是：

$$\text{taller} - \text{tall} + \text{short} = \text{shorter}$$

“taller”中减去“tall”的概念，加上“short”的概念，自然是“shorter”。

又或者：

$$\text{queen} - \text{king} = \text{woman} - \text{man}$$

看着很科幻，但这些案例是真实的。建议你亲自实验，发散思维尝试不同的单词运算。现在让我们思考一下，运算的结果为什么会和词义结果如此对应呢？

因为，和人类的语言一样，数学本身就是一门描述世界的语言。word2vec 中，向量对应了语言单词，运算规则对应了语言语法。区别于人类语言的语法设计，数学语言系统更注重数量的精确和逻辑运算的严谨，所以说它是一门更关注“精确”的语言。word 也好，对应的 vec 也好，它们只是对同一事物的不同描述，一个代表了我们眼中的世界，一个代表了计算机眼中的世界。

### 8.1.4 回顾

- 文本的 Embedding 实现：统计词频的文本向量。
- 单词的 Embedding：word2vec。
- 数学是一门语言，word 和 vec 是在不同语言空间下对同一个单词事物的描述。



## 8.2 输出序列的模型

我们任何人都逃脱不了记忆的捉弄。记忆会渐渐褪色，被遗忘后重新润色，会有新的色彩。

——《善良的男人》

8.1 节我们教会了计算机如何读懂单词的含义，本节，我们将尝试让计算机输出合理的文本序列。

### 8.2.1 RNN

Word-Embedding 帮助计算机获得描述单词文本的向量序列，假设我们希望计算机能够根据一篇文章的内容，预测文章的标题。那么显而易见，期望的输出同样也是一段文本序列，或者说是一段向量序列，而不再是一个独立的类别值。

可以将多次预测的输出串起来，形成序列。但是分类模型如 DNN 的每次输出之间都是相互独立的，下次预测和之前的预测之间没有相关性，如图 8-7 所示。

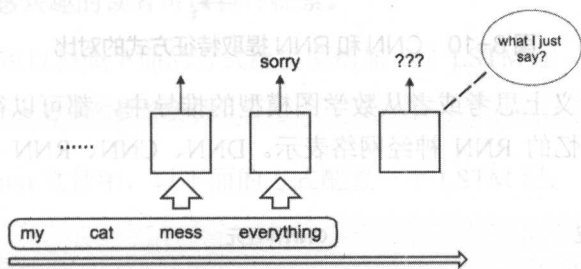


图 8-7 独立的预测模型

就像你说话的时候总要先思考之前说了些什么，再接着往下说。我们可以把每次预测的输出作为下一次的输入，让它们变得相关，如图 8-8 所示。

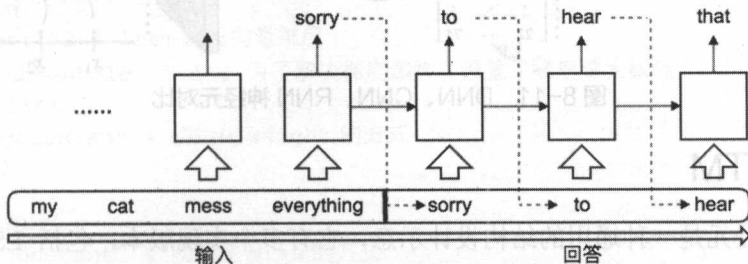


图 8-8 能够记忆状态的预测模型

为了让模型结构更简洁灵活一点，科学家们设计了一个能够随着输出序列流改变自

身状态的新神经元模型，这就是 RNN（Recurrent Neural Network）模型。如图 8-9 所示。

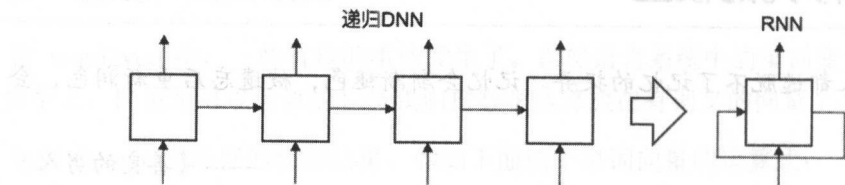


图 8-9 RNN 模型

RNN 的目标很简单，就是让模型随着输出流一起“变”。而它的灵感也很简单，如果说 CNN 的本质是在空间上关联参数，进而提取到空间特征；那么类似地，RNN 就是在时间上关联参数，提取序列的时间特征。如图 8-10 所示。

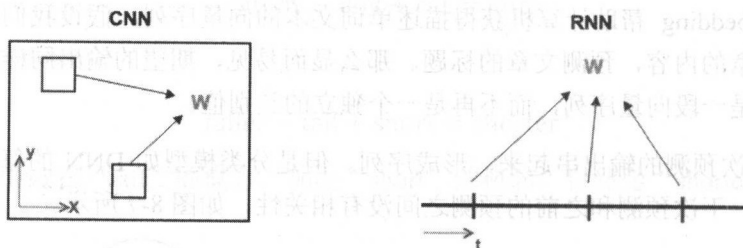


图 8-10 CNN 和 RNN 提取特征方式的对比

其实从记忆的意义思考或者从数学图模型的推导中，都可以得出一个结论：DNN 神经网络就是 0 阶记忆的 RNN 神经网络表示。DNN、CNN、RNN 神经元模型对比如图 8-11 所示。

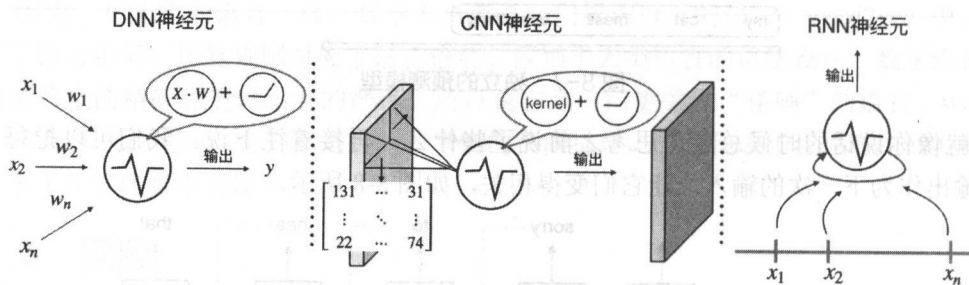


图 8-11 DNN、CNN、RNN 神经元对比

## 8.2.2 LSTM

RNN 神经元是一种通用的结构设计示意，它有多个实现版本，包括 LSTM 和 GRU。它们的结构基本一样，区别在于稍有差异的内核运算零件。

关于记忆这件事，对于很久远之前发生的事情，你可能印象深刻，犹如刚发生过，

而对于刚刚发生的事情，你可能很快就遗忘了——生物的记忆并不是随着时间线性变化的，而是非线性的。

LSTM 神经元内核正是在“记忆的管理”方向上做出优化的产物。

如图 8-12 所示，M 是记忆单元，X、Y、F 相当于三个 DNN 神经元，X 控制写入记忆的比例，Y 控制读取记忆的比例，F 决定遗忘的比例。让 X、Y、F 的神经元参数和 RNN 一起训练变化，这就是 LSTM。

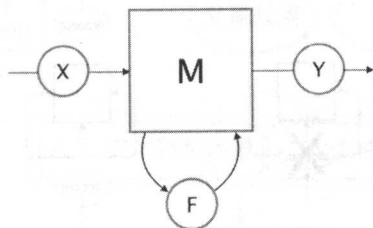


图 8-12 LSTM 模块示意图

关于 RNN 另一个实现版本是 GRU (Gated Recurrent Unit)，自 2014 年提出以来。它是简化版的 LSTM，感兴趣的读者可以自行探索。

在 Keras 中，你可以按照下面的方式给模型增加一个 LSTM 层。

```
model.add(LSTM(32)) # 输出向量长度 32
```

在 Caffe 的 prototxt 文件中，以下面的方式配置一个 LSTM 层：

```
layer {
  name: "lstm" # lstm 层
  type: "Lstm"
  bottom: "data"
  bottom: "xxx"
  top: "lstm"

  lstm_param {
    num_output: 32 # lstm 输出向量维度
    clipping_threshold: 0.1 # 为了解决梯度爆炸，设置了梯度最大阈值
    weight_filler {
      type: "gaussian" # 初始化 weight 的方式
      std: 0.1
    }
    bias_filler {
      type: "constant" # 初始化 bias 的方式
    }
  }
}
```

你可以找一些时序任务，如语音识别、NLP，Embedding 之后，输入到 LSTM 模型，达到你想要的目标。由于完成这些任务往往需要海量的数据支持，因本书篇幅有限，所以这里就不实现完整的项目了。

值得一提的是，去过拟合的 Dropout 要放在输入输出通道，而不要放在模型的循环记忆通道。LSTM 的 Dropout 如图 8-13 所示。

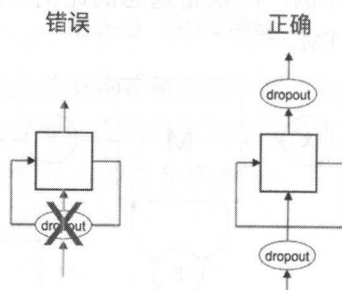


图 8-13 LSTM 的 Dropout

### 8.2.3 并用人工特征和深度学习特征——一个 NLP 模型的优化历程

对于一个 NLP（Nature Language Processing，自然语言处理）任务，举个例子：分类文本是散文—歌词—新闻。从零开始，一个优化模型的历程如下。

起初，用最简单、最快的方式人工构造出文本向量，完成分类。比如用 8.1 节提到的“词频统计特征”向量，如图 8-14 所示。分类模型可以用 DNN，也可以选择其他分类模型。

对待生命你不妨大胆冒险一点，  
因为好歹你要失去它。如果这世界上真有奇迹，  
那只是努力的另一个名字。生命中最难的阶段不  
是没有人懂你，而是你不懂你自己。

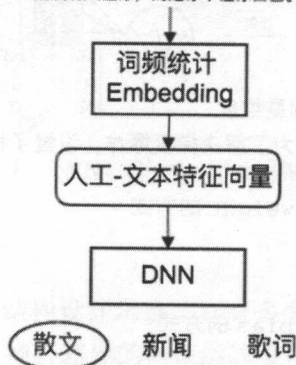


图 8-14 使用词频统计特征向量分类

接着，就是特征工程。将一些明显的特征直接构造出来，比如出处、作者等。注意中间的 merge 层，将文本特征向量和人工特征向量连接成一个新向量，输入到分类模型中。加入新特征如图 8-15 所示。

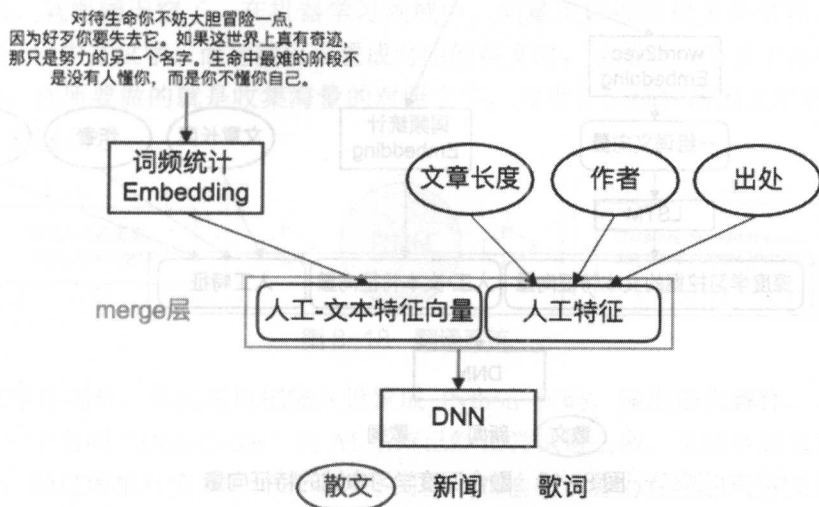


图 8-15 加入新特征

如果能比较好地完成上面两步，再调试优化一下模型参数，就应该是一个不错的模型了。但是文本中还有一些人工不易想出来的隐蔽特征，我们用深度学习模型自动学习出来，完成模型最后的提升。深层模型和传统机器学习模型融合的方式有两种：

- Ensample。
- 融合特征向量。

下面主要介绍后一种，使用线性融合 Ensample 请参考本书 2.6 节模型融合的实现思路。如图 8-16 所示，将 word2vec 生成的词义向量序列输入到 LSTM，生成描述整体文本序列的特征向量，把它和之前的向量连接在一起。

基本上优化一个 NLP 任务，都需经历特征工程 → 模型工程这样的历程。在深度学习和传统机器学习的接轨处，深度学习的价值就是挖掘人工挖不到的特征，提高模型成绩最后的 1%。

补充一点，NLP 中有时之所以需要传统机器学习而不使用纯粹的深度学习，一种解释是：虽然语言系统的语义有通用性，但 word2vec 本身还是从“别的数据集”中学习到语义模型，或者说 word2vec 是非监督的，所以在一个与训练 word2vec 数据集不同的数据任务中，word2vec 生成的描述词义的特征向量是有信息损失的。而且对于一些分类任



务，一些明显的人工特征对分类帮助很明显，比如上面的人工特征作者、出处等。

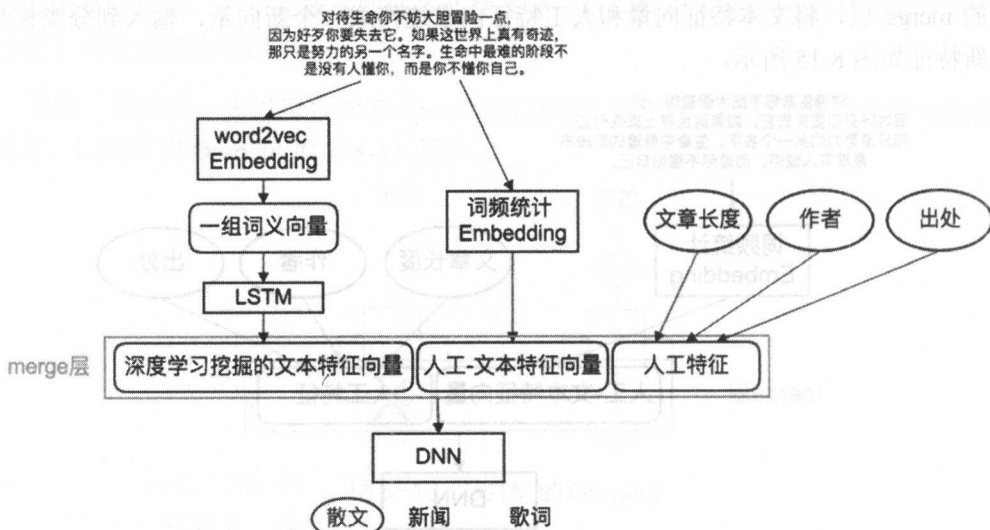


图 8-16 融合深度学习申城的特征向量

在运用深度学习时，我们要合理思考其作用，不要盲目迷信前沿科技。

## 8.2.4 反思：让模型拥有不同的能力

上文简单介绍了 RNN 输出文本序列的能力，其中一些有趣的事情值得反思，假设抛开我们的世界观，进入计算机的世界观，那么它是如何理解自己做了什么呢？

对于计算机而言，它并不懂单词、语言这些概念。它觉得自己所做的就是读取一个数值向量序列，按照代码的拟合规则（训练好的模型），输出一段数值序列——也就是序列到序列的映射。在计算机的世界观里，RNN 模型分析了输入序列的特征描述，并将这些描述信息映射到输出序列。这才是 RNN 模型在信息层面的根本意义。如图 8-17 所示。

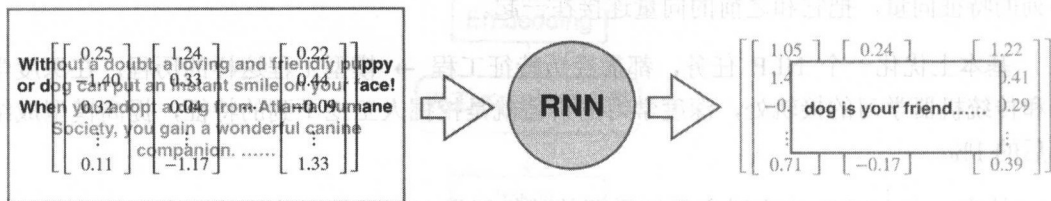


图 8-17 RNN 模型

这个理解有什么用呢？它告诉我们，RNN 只是在做样本的格式转换，输入输出格式都是向量序列，至于上层的应用意义可以由你定制。由于向量本身是长度为 1 的向量序列，

因此 RNN 的映射能力可以描述为:

序列 (or 向量)  $\rightarrow$  序列 (or 向量)

接下来,就海阔天空了。在机器学习领域中,向量可以描述很多种事物。当你把输入文字换成一段中文汉字,而把输出设置成对照的英文时,这就是一个基于深度学习的英文翻译系统。你所要做的就是收集海量的对照文字,并设计训练好两国文字的 word2vec 矩阵。如图 8-18 所示。



图 8-18 翻译系统

除了文字序列外,你还可以把输入设置成 Python 代码,输出语言解释——让计算机读懂代码,一个名叫“DeepCoder”的 AI 系统已经在尝试这么做。又或者想象模型的输入变成了声音,经过傅里叶变换采样的音序列值,而输出设置为对应的语言文字,那么这就是一个语音识别系统。如图 8-19 所示。

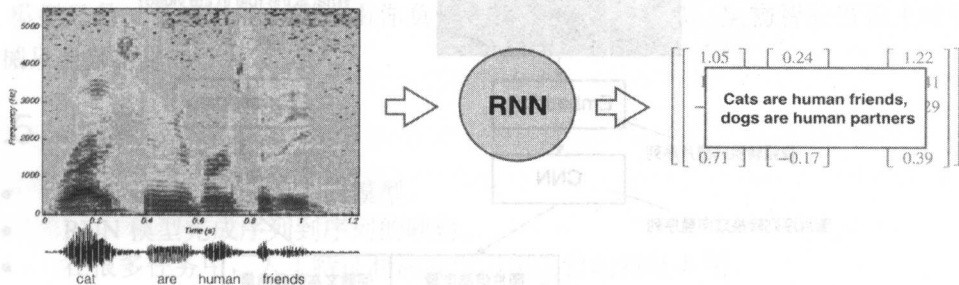


图 8-19 语音识别系统

在 CNN 章节中,我们通过 CNN 模型让计算机“看到”图片,将图片转换成一个包含图片内容的向量。那么如果把 CNN 输出的图片向量作为 RNN 模型层的输入,输出则是一段目标意义的文本序列,可以是标题、评价、打分,等等。那么这就是一个图文模型,可以做看图答题、颜值打分等。看图说话系统如图 8-20 所示。

视频是图片的序列,所以自然可以输入一段视频,生成语言文字。其实可以扩展一点,让视频本身携带一些问题,就像电视娱乐节目中的“视频问答”节目:给出一段视频,让你选择 A、B、C、D。要想让模型能够解决这类问题,需要融合 RNN 处理问题文本的能力和 CNN 处理图片的能力,最后用 DNN 在四个答案选项中做出选择。如图 8-21 所示。

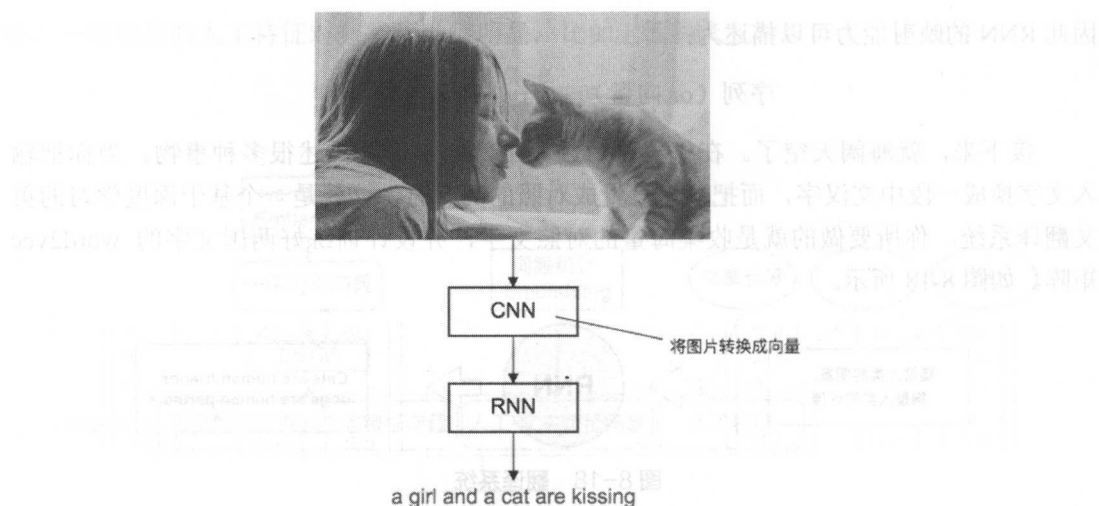


图 8-20 看图说话系统

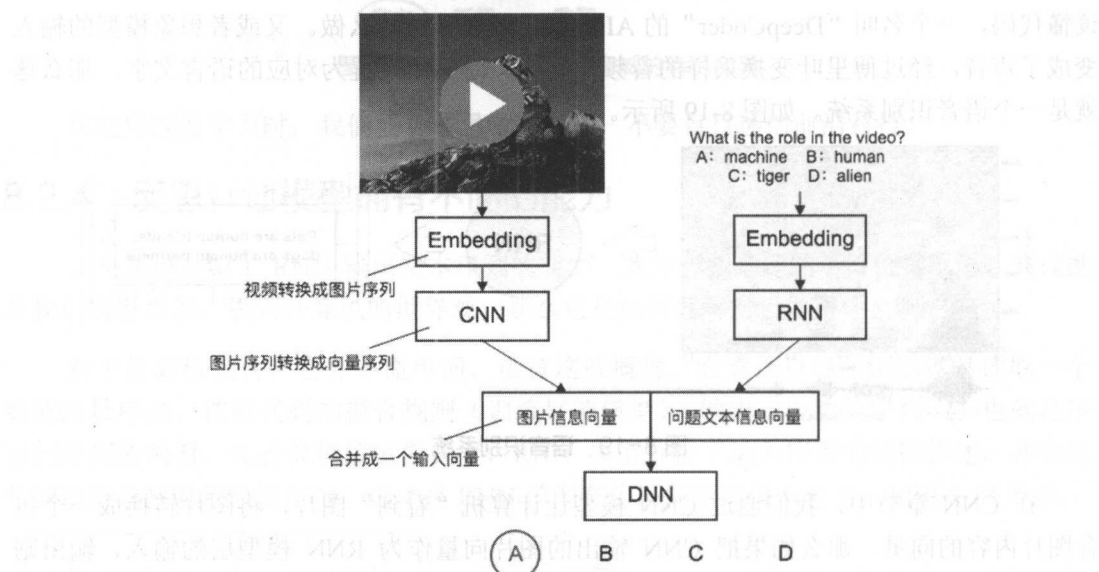


图 8-21 看图做题系统

还可以学习一些更抽象的东西——只要你的数据和模型深度能够支持，并且计算机足够强大。比如，我们希望模型能够嗅出文章的“文笔”，可以把相同作者的文章和不同作者的文章作为序列输入，输出一个“风格”意义的向量，在这之上完成分类识别任务。如图 8-22 所示。

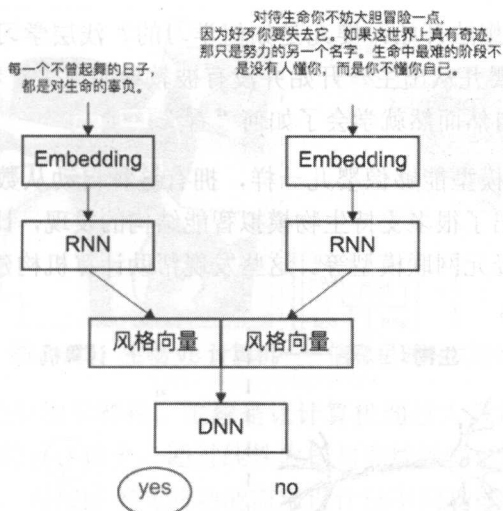


图 8-22 辨别系统——是不是同一个作者

上面所说的系统有些是真实存在的，但完成这样级别的模型需要海量的数据支持、昂贵的集群建设以及处理很多工程细节。

模型只是一个空白的大脑，由你负责训练它数据的灵魂。生物智能擅长主动的思考，而机械只会自动化的计算。

### 8.2.5 回顾

- RNN 是提取时间特征的模型。
- RNN 模型完成序列到序列的映射。
- 在很多任务中，人工特征和深度学习出的自动特征并用。

## 8.3 深度学习：原理篇总结

以前，这个世界只有黑暗。现在，是光明占了上风。

——True Detective Season

本节简单串一下第 4 至第 8 章的知识点，作为深度学习原理篇的结束。

### 8.3.1 原理小结

我们更关注深度学习模型的生物解释。

起点从一个强大的想法开始：婴儿是如何学习的？浅层学习对样本都需要量化的特征值，才能识别。然而婴儿从出生一开始并没有被教导如何去“看”，他（她）只是看了足够多的自然界样本，自然而然就学会了如何“看”。

我们希望计算机的模型能够像婴儿一样，拥有这种自动从数据中学习特征的能力。实验室的生物学研究给出了很多支持生物模拟智能结构的发现，比如神经元的刺激一响应函数特性、脑皮层的神经元网联模型等。这些发现帮助计算机构建模拟生理结构的神经网络模型。如图 8-23 所示。

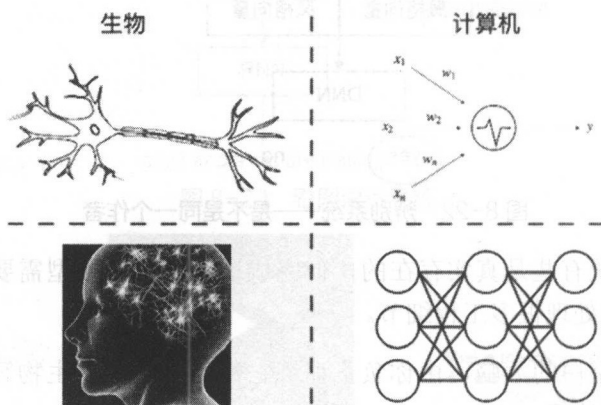


图 8-23 生物与计算机——模型对比

计算机和人类有着完全不同的生理基础。在生物系统中，外界的信息通过五感转换成生物刺激信号（spike），传递给大脑；而在计算机眼中，一切外界信息都用数值量化，组织成不同维度的数组或者序列，最终所有的事物信息可以通过一条向量描述。如图 8-24 所示。

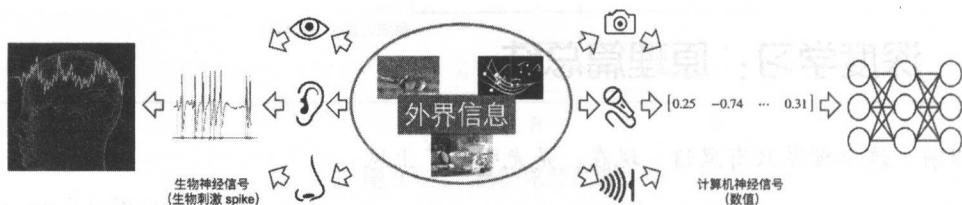


图 8-24 生物 vs 计算机——接收处理自然信号的方式对比

大脑皮层以结构完全一样的神经元，通过后天训练衍生出不同的皮层组织，处理对应的五官信号。与之相对的，数据既可以按照维度分类，也可以按照时空意义分类（空间维度数组和时间序列）。在神经元的网络模型中，通过更换神经元的内核，可以衍生出不同功能的层，模拟脑皮层不同的功能组织。如图 8-25 所示。



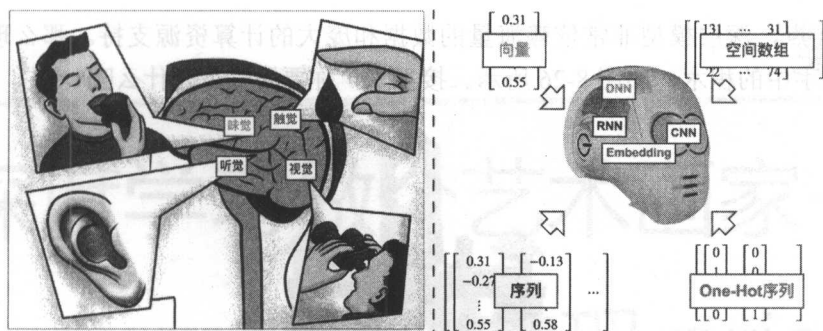


图 8-25 生物 vs 计算机——信息呈现形式对比

这就是深度学习的生物学解释。主题是让计算机通过大数据自动学习理解事物，代替人工特征解析。具体的设计理念：通过线性内核与非线性激活的组合，模拟适合计算机执行的人工神经元模型，再按照不同的功能需求设计出不同功能的神经元内核，并以层的方式，将它们融合起来训练。

### 8.3.2 使用建议

生物学能够帮助你理解深度学习模型的原理，但在工程中实际使用的时候，你需要切换视角。计算机并不懂我们输入的这些数值代表的应用意义，对于一个机器学习系统而言，它只看到两件事：输入输出的数据格式（I/O）以及模型处理信息的方式（代码）。

如表 8-1 所示，每个层模型做的事情无非是提取样本的描述，并映射到输出格式上。在工程应用编程中，每个模型层就好比一块积木，程序员只需按照输入输出的数据格式，将积木按接口形状拼接起来，构造出自己期望的模型即可。

表 8-1 各模型层数据格式及特征提取方式汇总

模 型	输入—输出	信息提取方式
DNN	向量—向量/数值	提取分类/回归意义的特征描述
CNN	(1、2、3 维) 空间数组—向量	提取数据空间形态的特征描述
Embedding	One-Hot 向量—向量	从 One-Hot 编码空间向其他意义空间的映射
RNN	序列—序列/向量	提取数据时间形态的特征描述

另外，所有的深层模型从来不局限于是某类特定任务，高端玩家既可以让 LSTM 代替 CNN 处理图像，也可以让 CNN 提取序列的特征。因为图像既可以按空间像素方式理解，也可以按时间方式理解（想象作画的过程）。只要数据格式能够对上，并且模型原理上解释得通，你就可以自由尝试，拿结果验证。

记住一点，深层模型非常依赖海量的数据和庞大的计算资源支持。那么现在，这些模型层是你手中的积木，如图 8-26 所示。接下来，你要用它搭建什么呢？

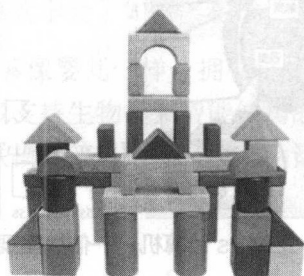


图 8-26 积木

# 用深度学习做个艺术画家—— 模仿实现 prisma

本章将探索深度学习落地到图像处理领域的方案，本章适合有一定深度学习实践经验的读者进阶阅读。

使用深度学习作画的起源是有三个德国研究员想把计算机调教成梵高，他们研发了一种算法，模拟人类视觉的处理方式。具体是通过训练多层卷积神经网络，让计算机识别，并学会梵高的“风格”，然后将任何一张普通的照片变成梵高的《星空》。

后来他们开创了 Deep Art 公司，在 Deep Art 公司，负责绘画的程序员是卷积神经网络（CNN）。输入一个艺术作品，比如梵高的《星空》，卷积神经网络就会自动提取出这幅画作的“风格特征”，并转换成风格模板保存下来。如图 9-1 所示。也就是说，卷积神经网络可以被看作一个机器艺术家。

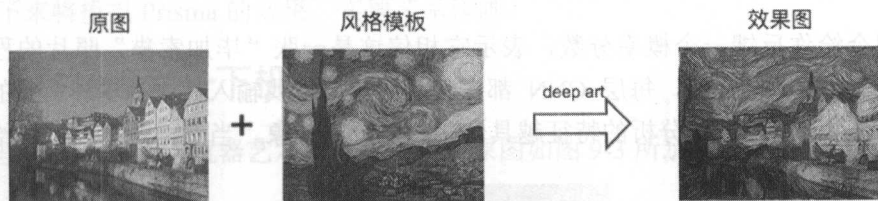


图 9-1 Deep art

Prisma 第一次将这项艺术作画技术成功商业化。Prisma 诞生于俄罗斯，是一个仅有 4 个年轻人历时一个半月开发出的图片处理应用。他们充分考虑了智能手机覆盖率的飞速增长，并且细致研究了用户行为。Prisma 接入的是以亿数量级的市场，俄国总统梅德韦杰夫也成为了 Prisma 的用户，他在 Instagram 上晒出了一张 Prisma 作品，迅速获得 8.7 万个赞。

Google 的 Deep Dream 也是一个会画画的计算机。它能够自动识别图像，筛选其中一些部分，进行夸张，以创造出一种迷幻效果。Deep Dream 完全开源，在几个主流的深度学习库如 Keras、Caffe 的官方 example 中，都有 Deep Dream 的实现 Demo。

本章我们将探索实现类似 Prisma 的效果。

说明：

本章完整项目地址：[https://github.com/bbfamily/prisma\\_abu](https://github.com/bbfamily/prisma_abu)

本项目演示视频：[m.v.qq.com/play.html?&vid=v0397sv1fab](https://m.v.qq.com/play.html?&vid=v0397sv1fab)，也可以在公众号 abu\_quant 中直接观看视频。

问题反馈：请在公众号 abu\_quant 给我们留言。

## 9.1 机器学习初探艺术作画

好的艺术家模仿皮毛，伟大的艺术家窃取灵魂。

——毕加索

本节介绍机器学习作画的简单原理，并展示输出效果。

### 9.1.1 艺术作画概念基础

第 6 章介绍了 CNN 如何提取图片中的图形特征，进而识别图片实物。现在，假设这里已经训练好了一个“识别毕加索绘制的猫”的深层卷积神经网络模型，如果把一张完全不同的照片输入模型，比如一张狗的照片，会发生什么呢？

模型会给你反馈一个概率分数，表示它相信这是一张“毕加索猫”照片的程度。这中间经历了很多 CNN 层，每层 CNN 都在狗狗照片上寻找输入样本是毕加索猫的图形特征证据，越底层的神经元分析的特征越具体，越高层越抽象。当然，最后模型会给出很低的分数。

图 9-2 是在狗照上识别毕加索猫的过程中，如果让模型能够修改输入的样本又会怎样呢？

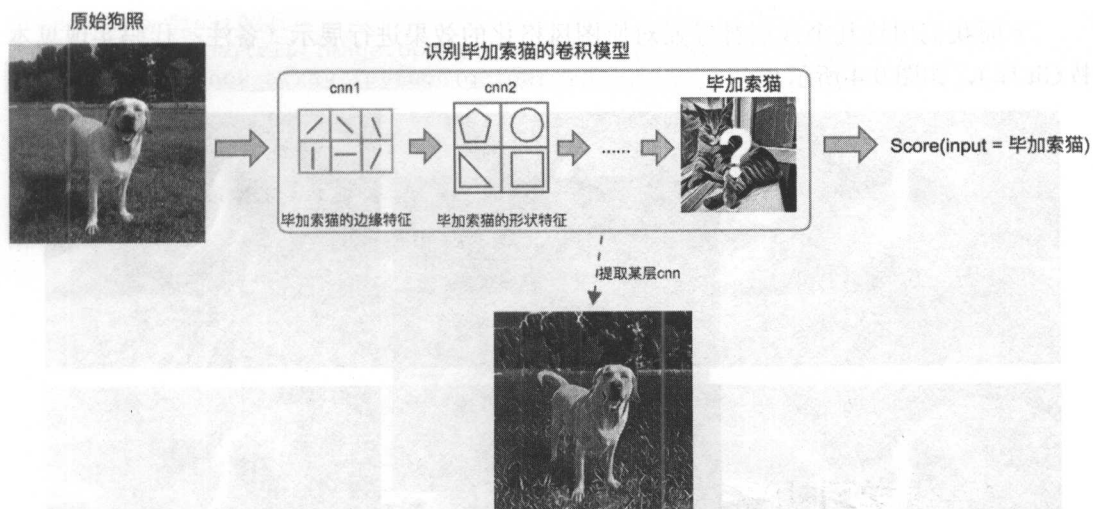


图 9-2 CNN 反馈修改输入图片

给模型网络中加一个反馈回路，让每一层网络可以朝着使最后分数变大的方向上修改狗狗照片。每次迭代网络中的每层都会在狗照上增加一些毕加索猫的特征痕迹，可以迭代很多次，让狗狗照片中加入越来越多的毕加索猫的实物特征。

这就是使用卷积神经网络艺术作画的概念基础，让艺术风格模型的 CNN 按图形特征修改输入图片，叠加艺术效果。大致的实现思路如下：

- (1) 输入特征图像，训练风格模型，让计算机学会艺术风格。
- (2) 输入待处理图，风格模型引导修改输入图片，生成新的图像，输出“艺术画”。

接下来将模拟 Prisma 的效果，实现艺术作画。

### 9.1.2 直观感受一下机器艺术家

这里我们展示一下机器艺术作画的效果，原图如图 9-3 所示。

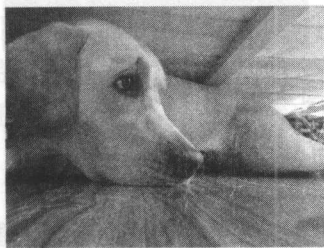


图 9-3 原图



下面我们用这几个浅层神经元对原图风格化的效果进行展示（备注：代码实现见本书 Git 库），如图 9-4 所示。

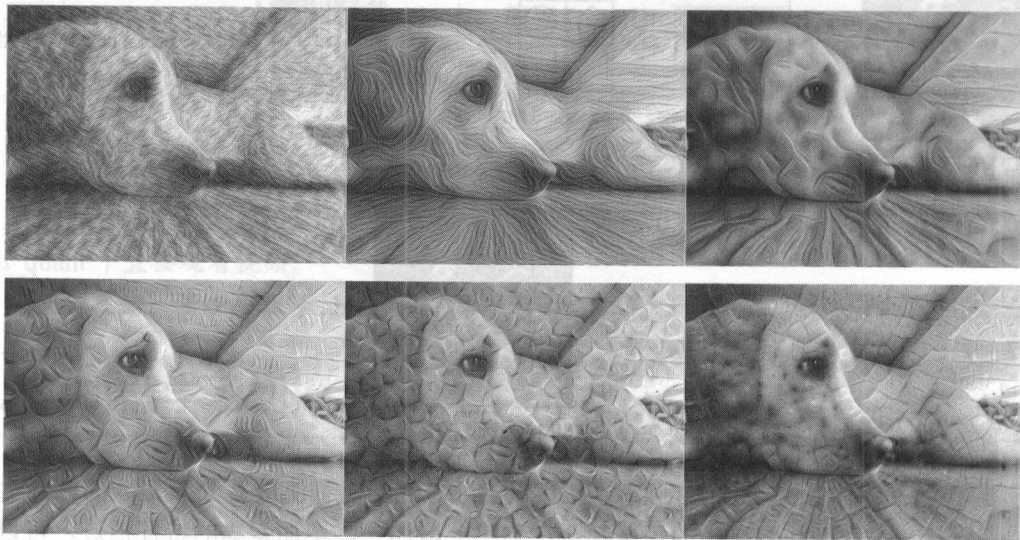


图 9-4 机器风格作画效果展示

有没有感觉到特征的识别由浅入深的一步一步增强，也就是从 edge，到 shape，再到复杂的 shape 循序渐进的过程，这里主要是卷积层把每层的特质放大进行夸张凸显。

图 9-5 展示了更多其他风格作画。



图 9-5 机器艺术作画效果图

### 9.1.3 一个有意思的实验

如果用 Prisma 做出一个图像，然后将它作为特征图像去引导新的图像生成会有什么效果呢？

```
guide = np.float32(
    tp.resize_img(PIL.Image.open('../prisma_gd/106480401.jpg'))
    PrismaHelper.show_array_ipython(guide)
```

输出如图 9-6 所示。



图 9-6 引导图

如图 9-7 所示，有些特征还是挖掘到了。

```
PrismaHelper.show_array_ipython(tp.fit_guide_img(s_file, gd_path,
    resize=True, size=640, iter_n=1500))
```

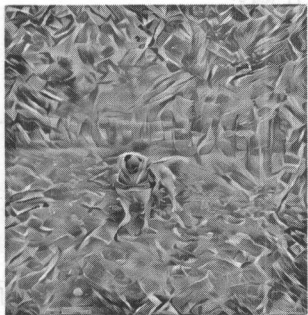


图 9-7 效果图

### 9.1.4 机器艺术作画的愿景

当机器能够根据图像的平面特征作画时，很多灵感也随之而来。比如我们可以从引导图中发现图像特征，从多个特征中寻找出存在的对象，并将这个特征融合到另一个图像中做特征融合。如果能够将特征识别融合做到极致，就可以完成如下假想场景。

- (1) 抬起头看到天边一朵云，看起来好像我家拉布拉多犬呢，是不是可以替换一下？
- (2) 用手机拍下这朵云，将狗狗的照片和云的照片发到云端进行特征识别融合。
- (3) 云端将融合后的图像发回给用户，如图 9-8 所示。



图 9-8 特征融合的愿景图

### 9.1.5 回顾

本节我们介绍了深度学习艺术作画的原理，并展示了直观的效果，在使用 Deep Dream 等开源项目实现上述效果时，速度非常缓慢，所以从 9.2 节开始我们将使用自己的方式实现快速 Prisma，实现秒级作画。

## 9.2 实现秒级艺术作画

天下武功，唯快不破。互联网竞争的利器就是快。

——雷军

和 Deep Art 相比，Prisma 的优势在于大大缩短了图像处理的时间，每张照片在 Prisma 系统内的处理时间控制在秒级别。而 Deep Art 更像是精工细作的手艺人，算法跑得虽然慢一些，但在细节表现力上更胜一筹。

在本书写作之前，笔者参考了几个艺术作画开源项目，都达不到真实 Prisma 的速度要求，本节将要使用的方式都是笔者原创的方法。首先笔者并不知道 Prisma 到底使用了什么方式使图像效果又好，速度又快，但是大概猜测的方向包括以下几种可能。

- (1) 大量的多 CPU、GPU 的机器（绝对不现实，成本根本无法控制）。
- (2) 未知的算法优化、网络框架优化（就算是这样，这也是我们没有能力突破的黑盒）。
- (3) 拥有很大的图像数据库，可以很快地检索出与输入图像相似度最高的图像，之

后相似特征提取、权重渲染。

(4) 针对图像的部分区域使用机器学习算法将特征层放大，配合一些图像处理技术，提升渲染速度。

第三种方式是说在拥有海量图片数据的前提下，作一个分类模型。对于一个输入图片，模型先分类出这是哪种类型，根据类型选择固定的特征提取方式进行渲染。而第四种方式是使用一些图片预处理技术减少机器学习算法的工作量。

由于我们没有足够的图片资源，而且第三点实现方式的速度瓶颈会在检索和相似度计算上，所以下面讲的内容是针对第四点展开试验的，可以在速度及渲染效果上都达到比较满意的效果。而唯一的缺陷就在适用性上，实际使用时需要调整一下参数，所以在实际使用中可以结合上述第三种方式，针对一定数量的样本作为训练集  $x$ ，对应的  $y$  是效果参数，对输入进行分类，再配合使用相似度等提高自动适配的能力。

## 9.2.1 主要实现思路分解讲解

下面还是使用 abul 这张图片作为输入：

```
from PrismaCaffe import CaffePrismaClass
import PrismaHelper
import numpy as np
import PIL.Image

abul_file = '../sample/abul.jpg'
cp = CaffePrismaClass(dog_mode=False)
PrismaHelper.show_array_ipython(
    np.float32(cp.resize_img(PIL.Image.open(abul_file))))
```

输出如图 9-9 所示。

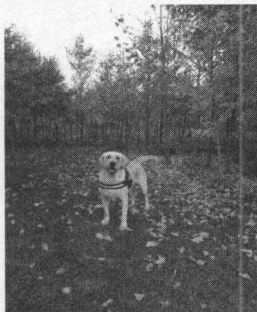


图 9-9 原始图

下一步，挑一张引导特征图像：

```
gd_path = '../prisma_gd/tooopen_sy_127260228921.jpg'
guide_img = np.float32(
    cp.resize_img(PIL.Image.open(gd_path), base_width=480,
        keep_size=False))
PrismaHelper.show_array_ipython(guide_img)
```

输出如图 9-10 所示。



图 9-10 引导图

如下所示，首先将图像转化为单通道，用 otsu 寻找 mask，通过 mask 确定 border 和 edges。

说明：otsu（大津算法，自适应阈值）。

- 关于 skimage otsu 等使用请参考：<http://scikit-image.org/>
- 关于 scipy ndimage 等使用请参考：<https://docs.scipy.org>

输入：

```
r_img = cp.resize_img(PIL.Image.open(abu1_file), base_width=480,
    keep_size=False)
# rgb 转化为单通道灰阶图像
l_img = np.float32(r_img.convert('L'))
# filters.threshold_otsu 需要 (-1, 1) 之间
l_img = np.float32(l_img / 255)
r_img = np.float32(r_img)

# 找出大于 otsu 的阈值作为 mask，需找 border
mask = l_img > filters.threshold_otsu(l_img)
# 不是适用所有图像都要 clear border，比如图像主题大部分需要保留时就不需要
clean_border = segmentation.clear_border(mask).astype(np.int)
coins_edges = segmentation.mark_boundaries(l_img, clean_border)
# 将值再次转换到 0-255
clean_border_img = np.float32(clean_border * 255)
clean_border_img = np.uint8(np.clip(clean_border_img, 0, 255))
```



```
PrismaHelper.show_array_ipython(clean_border_img)
```

输出如图 9-11 所示。

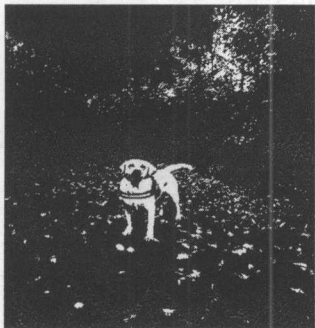


图 9-11 单通道图

目标就是只想摘取狗狗的图像，其他的都认为是噪音，可以使用 `ndimage.binary_opening`。达到效果了吗，试试看：

```
clean_border_img = ndimage.binary_opening(
    np.float32(clean_border_img / 255),
    structure=np.ones((5, 5)).astype(np.int)
)
clean_border_img = ndimage.binary_opening(clean_border_img).astype(
    np.int
)
PrismaHelper.show_array_ipython(clean_border_img * 255)
```

输出如图 9-12 所示。

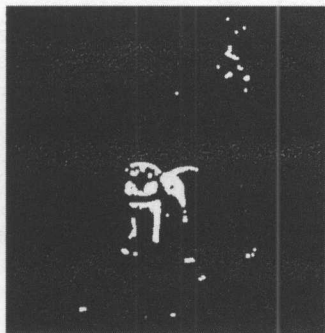


图 9-12 效果图

效果其实不太好，`ndimage.binary_opening` 效果与 CNN 中的最小池化层相似，目的就是击掉图中小物件，下面通过自定义简单卷积核来过滤，实现我们的需求。

代码如下：

```
# 最小的卷积核目的是保留大体图像结构
n = 5
small_window = np.ones((n, n))
small_window /= np.sum(small_window)
clean_border_small = convolve2d(clean_border_img, small_window,
                                mode="same", boundary="fill")

# 中号的卷积核是为了保留图像的内嵌部分，这里的作用就是狗狗的黑鼻子和嘴那部分
n = 25
median_window = np.ones((n, n))
median_window /= np.sum(median_window)
clean_border_convd_median = \
    convolve2d(clean_border_img, median_window, mode="same",
               boundary="fill")

# 最大号的卷积核，只是为了去除散落的边缘，很多时候没有必要，影响速度和效果
n = 180
big_window = np.ones((n, n))
big_window /= np.sum(big_window)
clean_border_convd_big = convolve2d(clean_border_img, big_window,
                                    mode="same", boundary="fill")

l_imgs = []
for d in range(3):
    # 分别对三个通道进行滤波
    rd_img = r_img[:, :, d]
    gd_img = guide_img[:, :, d]
    # 符合保留条件的使用原始图像，否则使用特征图像
    d_img = np.where(np.logical_or(
        clean_border_convd_median > 5 * clean_border_convd_big.mean(),
        np.logical_and(clean_border_small > 0, clean_border_convd_big \
                       > 2 * clean_border_convd_big.mean()))),
        rd_img, gd_img)
    l_imgs.append(d_img)
img_cvt = np.stack(l_imgs, axis=2).astype("uint8")
# 对转换出的图像进行一次简单浅层特征放大
d_img = cp.fit_img(nbk='conv2/3x3_reduce', iter_n=10, img_np=img_cvt)
PrismaHelper.show_array_ipython(np.float32(d_img))
```

输出如图 9-13 所示。

代码并不多，主要思路如下：

- (1) 通过 `filters.threshold_otsu` 找出图像的 mask。
- (2) `segmentation.clear_border(mask)` 抽取图像 border、edges。
- (3) 使用三个卷积核对图像进行滤波处理，这里的三个卷积核的分工请看上面的代

码注释。这里的滤波就是引导特征和原始图像的权重分配。



图 9-13 最后效果图

卷积的意义简单理解就是加权叠加，针对输入的单位相应得到输出。为什么要用卷积呢？工程上理解其实就是为了效率。如果上面的代码从目的出发，知道要滤除什么样的像素点，保留什么样的像素点，将这些编程为计算操作，然后使用 for 循环针对每一个像素点，在一定范围内（卷积核大小）执行计算操作，最后 for 循环一步一步前进，其实也能得出结果，但是运算的时间复杂度将大出几个数量级。

### 1. 使用图像特征作为 mask

上面的方法是使用 otsu 寻找图像边缘作为 mask 的依据，下面使用 skimage 中的 corner\_peaks 抽取图像特征作为 mask：

```
def show_features(gd_file):
    r_img = cp.resize_img(PIL.Image.open(gd_file), base_width=480,
                           keep_size=False)
    l_img = np.float32(r_img.convert('L'))
    ll_img = np.float32(l_img / 255)

    coords = corner_peaks(corner_harris(ll_img), min_distance=5)
    coords_subpix = corner_subpix(ll_img, coords, window_size=25)

    plt.figure(figsize=(8, 8))
    plt.imshow(r_img, interpolation='nearest')
    plt.plot(coords_subpix[:, 1], coords_subpix[:, 0], '+r',
             markersize=15, mew=5)
    plt.plot(coords[:, 1], coords[:, 0], '.b', markersize=7)
    plt.axis('off')
    plt.show()

def find_features(gd_file=None, r_img=None, l_img=None, loop_factor=1,
                  show=False):
    if gd_file is not None:
        r_img = cp.resize_img(PIL.Image.open(gd_file), base_width=480,
                               keep_size=False)
```

```

l_img = np.float32(r_img.convert('L'))
l_img = np.float32(l_img / 255)

coords = corner_peaks(corner_harris(l_img), min_distance=5)
coords_subpix = corner_subpix(l_img, coords, window_size=25)

r_img_copy = np.zeros_like(l_img)
rd_img = np.float32(r_img)

r_img_copy[coords[:, 0], coords[:, 1]] = 1

f_loop = int(rd_img.shape[1] / 10 * loop_factor)
for _ in np.arange(0, f_loop):
    """
        放大特征点, 使用 loop_factor 来控制特征放大倍数
    """
    r_img_copy = ndimage.binary_dilation(r_img_copy).astype(
        r_img_copy.dtype)
    r_img_copy_ret = r_img_copy * 255

if show:
    r_img_copy_d = [rd_img[:, :, d] * r_img_copy for d in
                    range(3)]
    r_img_copy = np.stack(r_img_copy_d, axis=2)
    PrismaHelper.show_array_ipython(r_img_copy)
return r_img_copy_ret

```

显示抽取出的图像特征点:

```
show_features('../prisma_gd/71758PICxSa_1024.jpg')
```

原始图如图 9-14 所示。

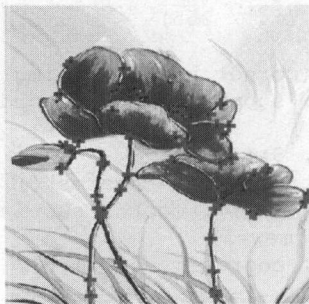


图 9-14 原始图

`find_features` 使用 `ndimage.binary_dilation` 来放大特征点, 使用 `loop_factor` 来控制特征放大倍数, 目的是结合引导特征做渲染时提升原始图像的特征权重, `find_features` 提取后的结果如下所示。



```
_ = find_features('../prisma_gd/71758PICxSa_1024.jpg', loop_factor=1,
                 show=True)
```

输出如图 9-15 所示。

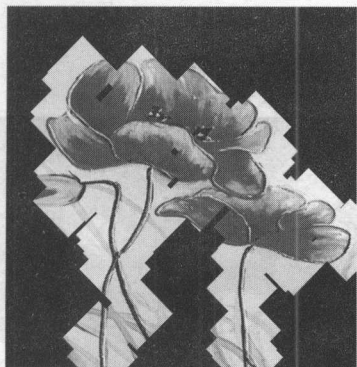


图 9-15 抽取效果图

下面用 IPython Notebook 的可交互形式更直观地看一下特征的抽取与放大。

```
from ipywidgets import interact

def find_features_interact(gd_file, loop_factor):
    r_img = cp.resize_img(PIL.Image.open(gd_file), base_width=480,
                          keep_size=False)
    l_img = np.float32(r_img.convert('L'))
    l_img = np.float32(l_img / 255)

    coords = corner_peaks(corner_harris(l_img), min_distance=5)
    coords_subpix = corner_subpix(l_img, coords, window_size=25)

    r_img_copy = np.zeros_like(l_img)
    rd_img = np.float32(r_img)

    r_img_copy[coords[:, 0], coords[:, 1]] = 1

    f_loop = int(rd_img.shape[1] / 10 * loop_factor)
    for _ in np.arange(0, f_loop):
        """
        放大特征点，使用 loop_factor 来控制特征放大倍数
        """
        r_img_copy = ndimage.binary_dilation(r_img_copy).astype(
            r_img_copy.dtype)
    r_img_copy_ret = r_img_copy * 255

    r_img_copy_d = [rd_img[:, :, d] * r_img_copy for d in range(3)]
    r_img_copy = np.stack(r_img_copy_d, axis=2)
    PrismaHelper.show_array_ipython(r_img_copy)
```



```
gd_file = ('../prisma_gd/71758PICxSa_1024.jpg', '../prisma_gd/st.jpg',
          '../prisma_gd/g1.jpg', '../prisma_gd/31K58PICSuH.jpg')
loop_factor = (0, 2, 0.1)
interact(find_features_interact, gd_file=gd_file,
         loop_factor=loop_factor)
```

输出如图 9-16 所示。

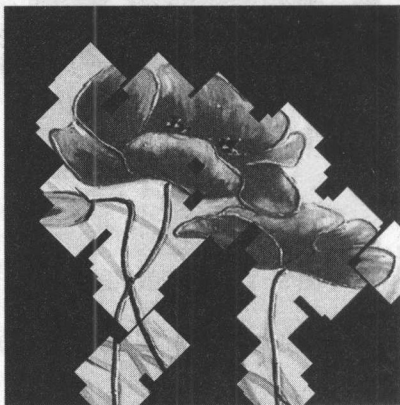


图 9-16 抽取效果图

## 9.2.2 使用统计参数期望与标准差寻找 mask

mask 的抽取方法可以有很多种方式，比如下面这个股票量化统计中经常使用的均值回复分析，用图像的均值和标准差来做滤波器，提取图像的 mask。

下面重构一下代码，分别使用三种滤波图像的方式生成 mask 查看效果。

(1) def do\_otsu(r\_img, l\_img, cb), 代码封装。

(2) def do\_features(r\_img, l\_img, cb, loop\_factor=1.0), 前面介绍的抽取图像点集特征放大的方式。

(3) def do\_stdmean(r\_img, l\_img, cb, std\_factor=1.0), 用均值标准差统计方式抽取 mask。

### 1. 使用多种方式 Prisma 图片

输入：

```
def do_otsu(r_img, l_img, cb, dd=True):
    mask = l_img > filters.threshold_otsu(
        l_img) if dd else l_img < filters.threshold_otsu(l_img)
    clean_border = mask
```

```

if cb:
    clean_border = segmentation.clear_border(mask).astype(np.int)
    clean_border_img = np.float32(clean_border * 255)
    clean_border_img = np.uint8(np.clip(clean_border_img, 0, 255))
    return clean_border_img

def do_features(r_img, l_img, cb, loop_factor=1.0):
    mask = find_features(r_img=r_img, l_img=l_img,
                        loop_factor=loop_factor)
    clean_border = mask
    if cb:
        clean_border = segmentation.clear_border(mask).astype(np.int)
        clean_border_img = np.float32(clean_border * 255)
        clean_border_img = np.uint8(np.clip(clean_border_img, 0, 255))
        return clean_border_img

"""
    感受一下图像滤波的部分
"""

def do_stdmean(r_img, l_img, cb, std_factor=1.0):
    mean_img = l_img.mean()
    std_img = l_img.std()

    mask1 = l_img > mean_img + (std_img * std_factor)
    mask2 = l_img < mean_img - (std_img * std_factor)

    clean_border = mask1
    if cb:
        clean_border = segmentation.clear_border(mask1).astype(np.int)
        clean_border_img1 = np.float32(clean_border * 255)
        clean_border_img1 = np.uint8(np.clip(clean_border_img1, 0, 255))

    clean_border = mask2
    if cb:
        clean_border = segmentation.clear_border(mask2).astype(np.int)
        clean_border_img2 = np.float32(clean_border * 255)
        clean_border_img2 = np.uint8(np.clip(clean_border_img2, 0, 255))

    # 上下两部分组合
    clean_border_img = clean_border_img1 + clean_border_img2
    clean_border_img = np.uint8(np.clip(clean_border_img, 0, 255))

    return clean_border_img

"""

```

将多个mask func用“与”的形式组合成mask滤波器

```
exp: tgt_mask_func = partial(together_mask_func,
```

```

func_list=[do_otsu, mask_stdmean_func, mask_features_func])
"""

def together_mask_func(r_img, l_img, cb, func_list):
    clean_border_img = None
    for func in func_list:
        if not callable(func):
            raise TypeError("together_mask_func must a func!!!")
        border_img = func(r_img, l_img, cb)
        if clean_border_img is None:
            clean_border_img = border_img
        else:
            clean_border_img = clean_border_img + border_img
    clean_border_img = np.uint8(np.clip(clean_border_img, 0, 255))
    return clean_border_img

"""
    使用 partial 统一 mask 函数接口形式
"""
mask_stdmean_func = partial(do_stdmean, std_factor=1.0)
mask_features_func = partial(do_features, loop_factor=0.88)

def do_conv_d_filter(n1, n2, n3, rb_rate, r_img, guide_img,
                    clean_border_img, convd_median_factor,
                    convd_big_factor):
    n = n1
    small_window = np.ones((n, n))
    small_window /= np.sum(small_window)
    clean_border_small = convolve2d(clean_border_img, small_window,
                                    mode="same", boundary="fill")

    n = n2
    median_window = np.ones((n, n))
    median_window /= np.sum(median_window)
    clean_border_conv_d_median = \
        convolve2d(clean_border_img, median_window, mode="same",
                    boundary="fill")

    n = n3
    big_window = np.ones((n, n))
    big_window /= np.sum(big_window)
    clean_border_conv_d_big = \
        convolve2d(clean_border_img, big_window,
                    mode="same", boundary="fill")

    l_imgs = []
    for d in range(3):
        """
        针对 rgb 各个通道处理

```



```

"""
rd_img = r_img[:, :, d]
gd_img = guide_img[:, :, d]

wn = []
for _ in np.arange(0, rd_img.shape[1]):
    """
        二项式概率分布
    """
    wn.append(np.random.binomial(1, rb_rate, rd_img.shape[0]))
if rb_rate <= 1:
    """
        针对 rgb 通道阶梯下降二项式概率
    """
    rb_rate = rb_rate - 0.1
w = np.stack(wn, axis=1)

d_img = np.where(np.logical_or(
    np.logical_and(
        clean_border_conv_d_median > conv_d_median_factor \
        * clean_border_conv_d_big.mean(), w == 1),
    np.logical_and(
        np.logical_and(clean_border_small > 0, w == 1),
        clean_border_conv_d_big > conv_d_big_factor \
        * clean_border_conv_d_big.mean()))),
    rd_img, gd_img)

l_imgs.append(d_img)
img_cvt = np.stack(l_imgs, axis=2).astype("uint8")
return img_cvt

def mix_mask_with_conv_d(do_mask_func, org_file=None, gd_file=None,
    nbk=None, enhance=None, n1=5, n2=38, n3=1,
    conv_d_median_factor=5.0, conv_d_big_factor=0.0,
    cb=False, rb_rate=1, r_img=None,
    guide_img=None, all_mask=False, show=False):
    if r_img is None:
        r_img = cv2.resize_img(PIL.Image.open(org_file),
            base_width=480, keep_size=False)

    l_img = np.float32(r_img.convert('L'))
    l_img = np.float32(l_img / 255)
    r_img = np.float32(r_img)

    if show:
        PrismaHelper.show_array_ipython(np.float32(r_img))

    if not callable(do_mask_func):
        raise TypeError(
            'mix_mask_with_conv_d must do_mask_func a func')

```

```

clean_border_img = np.ones_like(
    l_img) * 255 if all_mask else do_mask_func(r_img=r_img,
                                                l_img=l_img, cb=cb)

if show:
    PrismaHelper.show_array_ipython(np.float32(clean_border_img))

if guide_img is None:
    if gd_file is not None:
        guide_img = np.float32(
            cp.resize_img(PIL.Image.open(gd_file), base_width=480,
                           keep_size=False))
    else:
        guide_img = np.zeros_like(r_img)

img_cvt = do_convd_filter(n1, n2, n3, rb_rate, r_img, guide_img,
                          clean_border_img,
                          convd_median_factor=convd_median_factor,
                          convd_big_factor=convd_big_factor)

if nbk is not None:
    img_cvt = cp.fit_img(org_file, nbk=nbk, iter_n=10,
                        enhance=enhance, img_np=img_cvt)

if show:
    PrismaHelper.show_array_ipython(np.float32(img_cvt))
return img_cvt

```

使用特征 do\_features mask 方式, 注意 rb\_rate=0.66 的使用, 这里使用它的目的是使特征边缘平滑过渡到引导特征中, 当然这里还可以有各种优化方式, 比如向下调整 convd\_median\_factor, 使原始特征边缘提取更加圆润平滑。

```

_ = mix_mask_with_conv_d(partial(do_features, loop_factor=1.1),
    '../prisma_gd/71758PICxSa_1024.jpg',
    '../prisma_gd/cx6.jpg', 'conv2/3x3_reduce', rb_rate=0.66,
    show=True)

```

输出如图 9-17 所示。

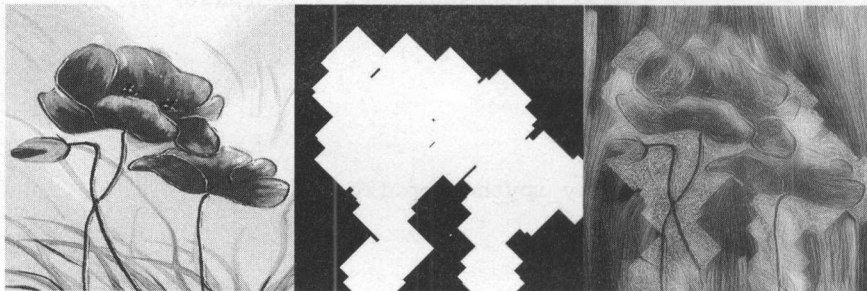


图 9-17 效果图



球员库里这张图使用均值回复 mask 方式可以对图片产生比较好的效果, 使用其他两种方式效果均不佳, 读者可以自行测试效果。

```
_ = mix_mask_with_convnd(mask_stdmean_func, '../sample/kl.jpg',
                          '../prisma_gd/cx7.jpg', 'conv2/3x3_reduce',
                          n2=88, convnd_median_factor=0.1, rb_rate=1,
                          show=True)
```

输出如图 9-18 所示。

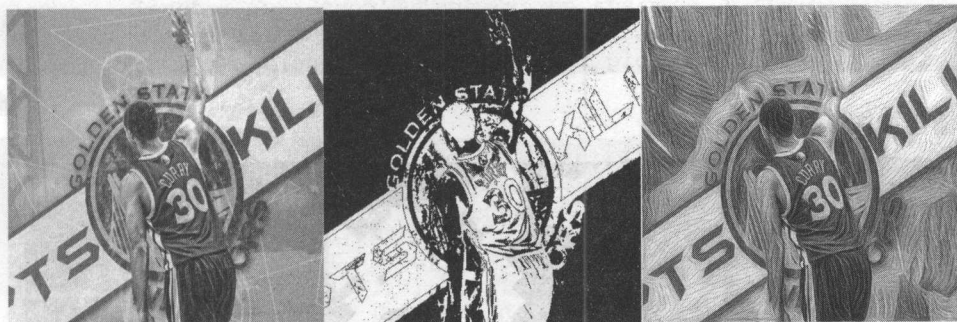


图 9-18 效果图

使用 abu2 看看现在这种方式的运行效率, %time 计算一下耗时。请注意参数, n3=1、convnd\_median\_factor=0.2、convnd\_big\_factor=0.0。也就是说, 不使用最大的卷积核, 速度会非常快, 只用了 5.62 s, 并且包含一些额外的代码工作, 如显示原图等。

```
% time _ = mix_mask_with_convnd(do_otsu, '../sample/abu2.jpg',
                                '../prisma_gd/k5.jpg', 'conv2/3x3_reduce',
                                n3=1, convnd_median_factor=0.2,
                                convnd_big_factor=0.0, show=True)
```

输出如图 9-19 所示。

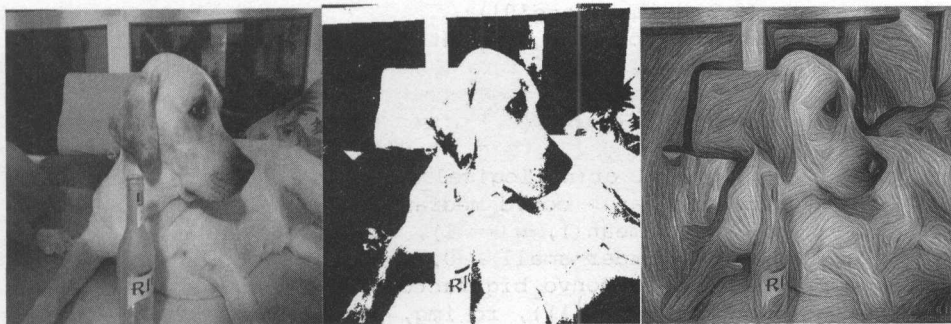


图 9-19 效果图

abu5 使用 partial(together\_mask\_func, func\_list=[do\_otsu, mask\_features\_func]), 组合多

个特征抽取 mask 函数，多个滤波函数以“与的关系”进行组合，对图像进行 mask。

这里如果不使用 `together_mask_func`，则单独每个都要再次调整一些参数。比如单独使用 `do_otsu`，就要调大 `n2` 核的大小，影响了速度，而 `mask` 函数合并特征完美快速实现了需求。

```
tgt_mask_func = partial(together_mask_func,
                        func_list=[do_otsu, mask_features_func])
_ = mix_mask_with_conv2d(tgt_mask_func, '../sample/abu3.jpg',
                        '../prisma_gd/cx3.jpg', 'conv2/3x3_reduce',
                        cb=False, n2=68, n3=1, convd_median_factor=1,
                        convd_big_factor=0.0, show=True)
```

输出如图 9-20 所示。

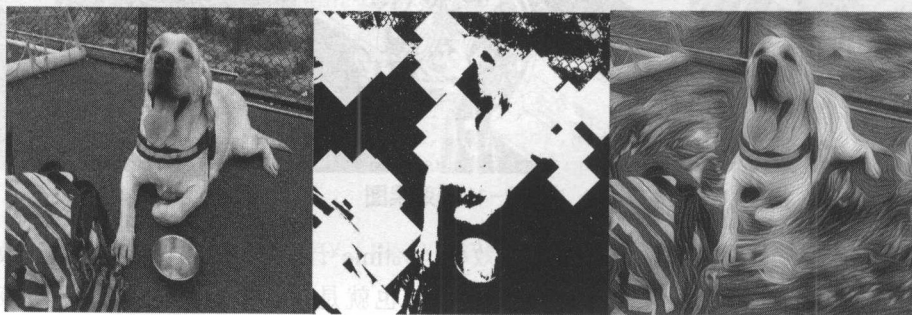


图 9-20 效果图

## 2. 配合使用预处理图像增强，随机 rgb 浅层 edges 等增强 Prisma 效果

下面对笔者的偶像艾弗森，使用图像预处理的方式来做图。首先用 `Contrast`，注意这里设置了参数 `rb_rate=0.88`，从代码来看它的作用是：

```
wn = []
for _ in np.arange(0, rd_img.shape[0]):
    wn.append(np.random.binomial(1, rb_rate, rd_img.shape[1]))
if rb_rate <= 1:
    rb_rate = rb_rate - 0.1
w = np.stack(wn, axis=1)

d_img = np.where(np.logical_or(np.logical_and(
    clean_border_conv2d_median > convd_median_factor * \
    clean_border_conv2d_big.mean(), w == 1), np.logical_and(
    np.logical_and(clean_border_small > 0, w == 1),
    clean_border_conv2d_big > convd_big_factor * \
    clean_border_conv2d_big.mean()))), rd_img, gd_img)
```

逻辑中 `np.logical_and` 添加 `w == 1` 的判断，这里使用二项式分布，增强渲染的迷幻效果，即随机在 `rgb` 某一个通道中渲染一下引导特征。

`rb_rate = rb_rate - 0.1` 的作用是3个通道的随机渲染概率阶梯下降, 这里也可以有其他各种渲染变种。

例如, 这里的 `w` 二项式分布矩阵就类似下面这个示例矩阵, 针对图像中每一个像素点 0.8 的概率为 1, 0.2 的概率为 0。

```
wn = []
for _ in np.arange(0, 20):
    wn.append(np.random.binomial(1, 0.8, 20))
w = np.stack(wn, axis=1)
w
```

输出:

```
array([[1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1],
       [1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0],
       [1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0],
       [1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1],
       [0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1],
       [1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1],
       [1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0],
       [1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1],
       [0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1],
       [1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1],
       [1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0],
       [1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0],
       [1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1],
       [1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1],
       [1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1],
       [0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1],
       [1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1]])
```

输入:

```
tgt_mask_func = partial(together_mask_func,
                        func_list=[do_otsu, mask_stdmean_func])
_ = mix_mask_with_convdd(tgt_mask_func, '../sample/lfs.jpg',
                        '../prisma_gd/cx11.jpg',
                        'conv2/norm2', enhance='Contrast',
                        rb_rate=0.88, n2=180, n3=1,
                        convd_median_factor=0.01,
                        convd_big_factor=0.0, show=True)
```

输出如图 9-21 所示。



图 9-21 效果图

感觉不是很帅，那怎么行，前置一个 Sharpness 预处理效果看看：

```
_ = mix_mask_with_convdo_otsu, '../sample/lfs.jpg',
    '../prisma_gd/cx11.jpg', 'conv2/norm2',
    enhance='Sharpness', rb_rate=0.88, cb=False,
    n2=188, n3=1, convd_median_factor=0.01,
    convd_big_factor=0.0, show=True)
```

输出如图 9-22 所示。

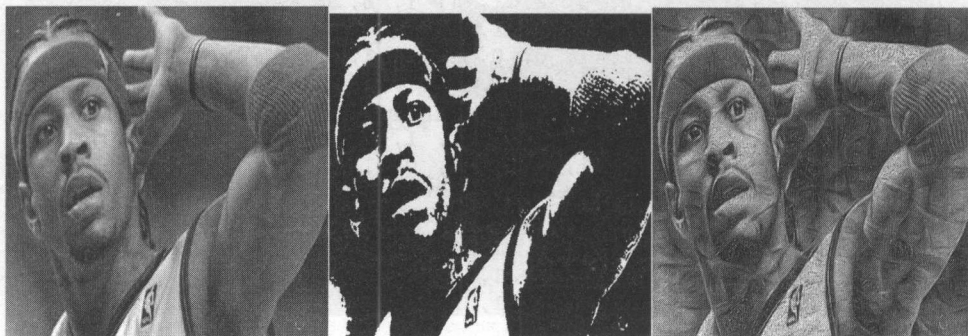


图 9-22 效果图

上面做出的两个效果，仔细观察就会发现，其实他们并没有使用引导图的 sharp 特征，只是通过阶梯 rgb 渲染，在原始图像上泼上了一层浅层的 edges 特征，这样实际上不需要使用上面这种实现方式。注意 mix\_mask\_with\_convdo 的参数 all\_mask，当 all\_mask 为 true 时，将整个图像的 mask 全设置 255，代码如下：

```
clean_border_img = np.ones_like(l_img) * 255 if all_mask else
do_mask_func(r_img=r_img, l_img=l_img, cb=cb)
```

所以如果想使用浅层特征 edges，直接设置 all\_mask 就可以了。



### 9.2.3 工程代码封装结构及使用示例

将上面的代码再次重构到文件 PrismaWorker 中, 代码详情请查阅 PrismaWorker.py。

```
from PrismaWorker import PrismaWorkerClass
pw = PrismaWorkerClass()
```

使用两个 GTA5 的图片, 融合摩托车大哥到大部队中。如图 9-23 所示:

```
pw.cp.resize_img(PIL.Image.open('../prisma_gd/gta2.jpg'),
                 base_width=480, keep_size=False)
```



图 9-23 原始图片

注意, `partial(do_otsu, dd=False)` 中的 `dd` 参数代表 `otsu` 后是取内部还是取反向的外部, 如下面的黑白 mask 图, `dd=False` 可以取到骑手, 否则将取到外部背景。

```
_ = pw.mix_mask_with_convdpartial(pw.do_otsu, dd=False),
    '../sample/gta4.jpg',
    '../prisma_gd/gta2.jpg',
    'conv2/3x3_reduce',
    enhance='Sharpness', n2=88, n3=1,
    convd_median_factor=1.5,
    convd_big_factor=0.0, show=True)
```

输出如图 9-24 所示。



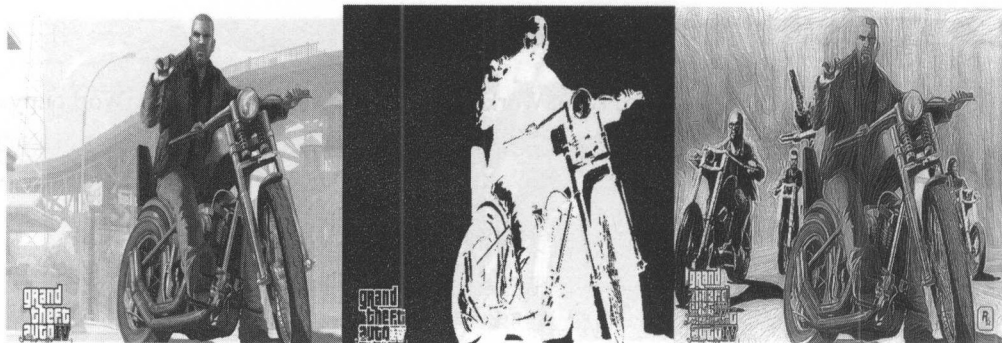


图 9-24 效果图

效果还算不错，除了左下角两个标准的重叠。

接下来使用批量处理引导图像，预处理、特征放大层等参数排列组合使用 PrismaMaster，详情请查询代码 PrismaMaster.py。如下代码所示，可以生成所有参数排列组合的输出结果。

```
import PrismaMaster

cb = False
n1 = 5
n2 = 38
n3 = 1
convd_median_factor = 0.6
convd_big_factor = 0.0
loop_factor = 1.0
std_factor = 0.88

nbk_list = filter(lambda nbk: nbk[-8:-1] <> '_split_',
                  cp.net.blobs.keys()[1:-2])[10]
org_file_list = ['../sample/bz1.jpg']
gd_file_list = [None, '../cx/cx3.jpg', '../cx/cx6.jpg',
                '../cx/cx7.jpg', '../cx/cx10.jpg']
enhance_list = [None, 'Sharpness', 'Contrast']
rb_rate_list = [0.85, 1.0]

save_dir = '../out/2016_11_24'
PrismaMaster.product_prisma(org_file_list, gd_file_list, nbk_list,
                             enhance_list, rb_rate_list, 'otsu_func',
                             n1, n2, n3, std_factor, loop_factor,
                             convd_median_factor,
                             convd_big_factor, cb, save_dir)
```

接下来再做一个 GUI 的可视化操作界面 PrismaController，使用了 traitsui 库，详情请查看 PrismaController.py。GUI 控制界面如图 9-25 所示。

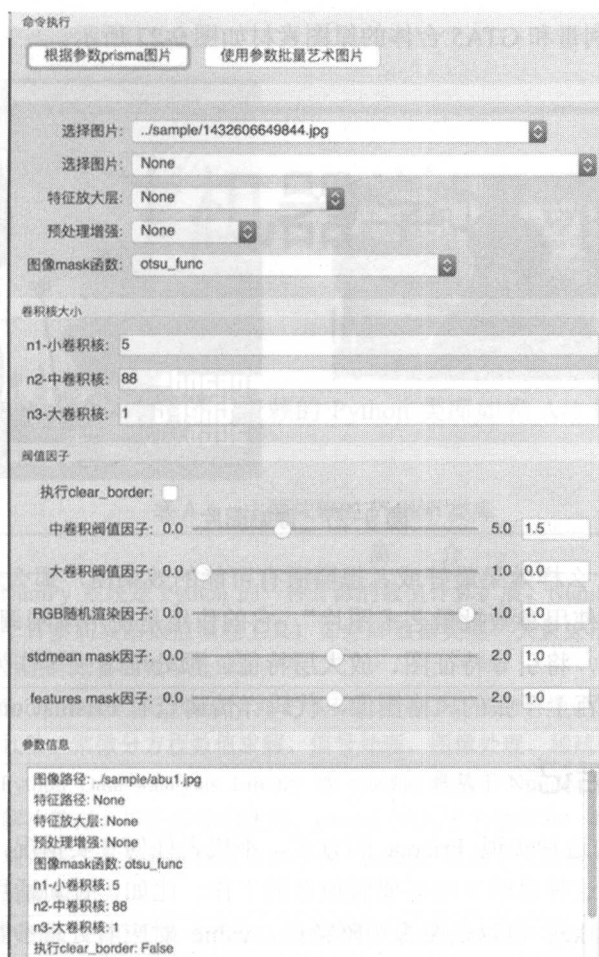


图 9-25 GUI 控制界面

基于这样一个方便微调的 GUI 下可以很方便地对图像进行微调，做出很多酷炫的图像，比如图 9-26 所示的做的两张基于 GTA 风格的图像。

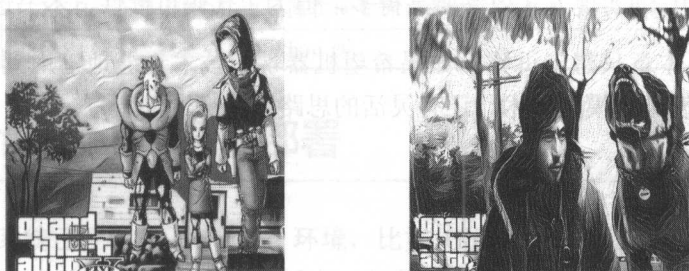


图 9-26 GTA 风格效果图

图 9-26 这张犀利哥和 GTA5 合体的原图素材如图 9-27 所示。

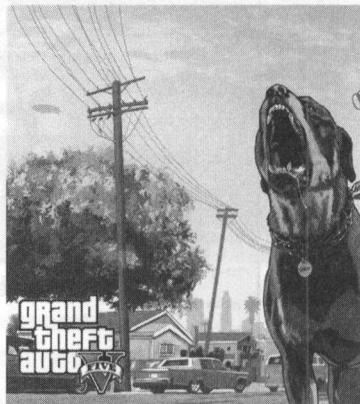


图 9-27 原始图片

如果你不知道什么样效果最好或者想要所有可能的效果图，那么你可以看到 GUI 的界面上还有个按钮“使用参数批量艺术图片”，它的作用是使用刚刚调整好的  $n1$ 、 $n2$ 、 $dd$  等参数作为固定参数，将引导特征图、放大层特征、预处理增强等作为所有可能的排列组合，通过一键生成成百上千张的风格图像，代码详情请查看 `PrismaController.py`。

## 9.2.4 回顾和后记

(1) 本节所讲的这种实现 Prisma 的方式，不代表任何真实情况，只是一种可能的技术实现思路，并且在这种思路下还需要做很多的工作。比如，针对适用性的问题也许要保存大量字典，字典的 key 可以是图像矩阵特征，value 对应着处理参数，然后针对输入的图像进行分类，或者根据特征相似度匹配来认定应该使用哪些参数等种种复杂问题需要处理。

(2) 本节的代码并没有过多关心运行效率等问题，比如针对图像保存。读取 `scipy.misc` 比用 PIL 的实现方式效率要高得多，但为了代码可读性，本书选择使用 PIL。

总的来说，本章只想告诉你，如果希望机器学习技术无缝地落地到某个领域时，需要的不仅仅是深度学习模型技术，还有灵活的思路以及变通的智慧。

# 机器学习环境部署

本书代码在 Python 2.7 环境下，依赖的 Python 类库如表 A-1 所示（深度学习框架见附录 B，这里不包含）。

表 A-1 主要依赖的 Python 类库

软件包名称	简介
NumPy	NumPy 系统是 Python 的一种开源的数值计算扩展。NumPy (Numeric Python) 提供了许多高级的数值编程工具，如矩阵数据类型、矢量处理，以及精密的运算库。专为进行严格的数字处理而产生
SciPy	SciPy 函数库在 NumPy 库的基础上增加了众多的科学计算的常用库函数。例如线性代数、常微分方程数值求解、信号处理、图像处理、稀疏矩阵等
pandas	Python Data Analysis Library 或 pandas 是基于 NumPy 的一种工具，该工具是为了解决数据分析任务而创建的。pandas 纳入了大量库和一些标准的数据模型，提供了高效操作大型数据集所需的工具。pandas 提供了大量能使我们快速便捷处理数据的函数和方法
Matplotlib	Python 2D 绘图领域使用最广泛的库。它能让使用者很轻松地将数据图形化
Seaborn	Matplotlib 封装，使得绘制更加简单
Scikit-Learn	Scikit-Learn 的基本功能主要被分为六个部分，分类、回归、聚类、数据降维、模型选择和数据预处理。Scikit-Learn 中的机器学习模型非常丰富，包括 SVM、决策树、GBDT、KNN 等，可以根据问题的类型选择合适的模型
XGBoost	XGBoost 是 GBDT 的一种实现类库，见本书“1.10 融合成群体”
OpenCV	OpenCV 是视觉处理 Python 库

## A.1 Anaconda 环境部署

很多操作系统已经内置了 Python 环境，比如 Ubuntu、CentOS、Mac OS，这些系统的很多功能都依赖于 Python 的某个版本，如果自己编写程序所使用的 Python 版本或 Python 库版本不一致时，就需要升级或者降级版本，在升级或降级后导致的不兼容问题数

不胜数。为了不污染系统运行的 Python 环境，在这里建议使用 anaconda 来管理开发的 Python 环境。

anaconda 所建立的 Python 环境与系统的 Python 环境是完全隔离的，而且 anaconda 还可以创建多套 Python 环境，这样就保证了开发环境和系统环境互相独立。除了 anaconda 外，还有 virtualenv 等流行的开发环境管理器。anaconda 的优势在于安装简单，且集成了几乎所有的科学计算库，同时支持 Linux、Mac os、Windows 主流平台。

anaconda 的下载地址是 <https://www.continuum.io/downloads>，读者可根据所使用的操作系统下载对应的版本。安装完成之后，就拥有了本书第 1 章所使用的大部分 Python 库。

## 1. Windows

双击 anaconda 安装程序，并按照提示安装到默认路径。

## 2. Mac OS

Mac OS 提供了两种安装程序，一种是 dmg 格式的安装程序，就是带图形化的版本，提供了图形化的安装和管理。但是图形化的管理程序经常出现卡死的情况，因此不建议使用。直接下载 sh 格式命令行的安装程序。

打开 Terminal.app 或者 iTerm2.app 并输入：

```
$ bash ~/Downloads/Anaconda2-4.2.0-MacOSX-x86_64.sh
```

按照提示信息使用默认选项进行安装，安装默认输出到~/anaconda2/下面。

## 3. Linux

下载 sh 格式的安装程序，和 Mac OS 的命令行安装过程一样，打开终端程序并输入：

```
$ bash ~/Downloads/Anaconda3-4.2.0-Linux-x86_64.sh
```

读者应以下载的文件名称替换上面的安装名称，最后注意添加 PATH：

```
export PATH=~/anaconda/bin:$PATH
```

## 4. 卸载方法

Windows 下的卸载和别的程序一样，都是在控制面板中卸载。MacOS 和 Linux 中的卸载则使用 `rm -rf ~/anaconda2/`即可。



## A.2 使用 Anaconda

conda 使用的最多的命令就是 install 命令：

```
conda install <包名>
```

conda install 的好处是提前分析出包的依赖关系，比如需要安装 a，它可以分析出 a 的安装需要升级现在环境中的 b，降级现在环境中的 c，这样以供用户来综合决策是否需要安装 a。当用户决定安装 a 的时候，conda 会将上述升级 b 与降级 c 的操作一并执行。

conda 也可以安装指定版本，例如：

```
conda install numpy=1.11.2
```

conda 卸载包使用 uninstall 命令：

```
conda uninstall <包名>
```

anaconda 的常用命令如表 A-2 所示。

表 A-2 Anaconda 常用命令

用 法	简 介
conda info	conda 基本信息，包括所在平台，版本，路径等
conda list [-n envName]	安装了的软件包
conda search packageName	搜索软件包
conda create envName	创建一个环境
conda install [-n envName] packageName	安装软件包
conda update [-n envName] packageName	更新软件包
conda remove [-n envName] packageName	删除软件包

conda 在安装之后会生成一个默认的环境 root，参数 -n 是开发环境的名称，省略 -n 参数就是对当前的环境执行对应的操作。

## A.3 pip 环境部署

对于 conda 中没有的安装包，如 xgboost，则可以使用 pip 工具配合安装。Python 2.7 内置 pip。Mac 用户也可以选择使用 Homebrew 补充缺失的安装包。

pip 使用 install 命令安装包：

```
pip install <包名>
```

可以指定版本，例如：

```
pip install xgboost==0.6
```

卸载使用 `uninstall` 命令：

```
pip uninstall <包名>
```

## A.4 安装 abupy

本书中使用了 `abupy` 类库的机器学习模块，读者可以在 <https://github.com/bbfamily/abu> 中下载并使用。如果下载地址有变动，则可以通过关注公众号 `abu_quant` 获取最新 Git 地址，代码具体部署方式请阅读 Git 项目首页，或者阅读公众号中的 `abu` 文档。

用 `pip` 安装 `abupy`，需在终端输入如下命令：

```
pip install abupy
```

使用 `pip` 也可以安装 `abupy`，但是没有 Git 上的相关例子及文档。

conda info	conda info
conda list	conda list
conda search	conda search
conda create	conda create
conda env	conda env
conda update	conda update
conda remove	conda remove

## 附录 B

# 深度学习环境部署

本书使用 Keras、Caffe 深度学习类库，其中 Keras 的安装依赖 TensorFlow。对于一些系统环境，Caffe 安装略复杂，新手可以先暂且跳过，仅仅阅读本书 Keras 相关的部分，一定程度之后再选择 Caffe 作为进阶的工具。

安装前提：请确认安装好附录 A 的 Python 类库，如 NumPy、OpenCV 等。

## B.1 部署 Keras

Keras 支持 TensorFlow 和 Theano 作为后端引擎，推荐使用 TensorFlow 作为后端。Keras 的主要依赖包：

- SciPy 科学计算库。
- Pyyaml。
- HDF5、h5py 可选，一种为存储和处理大容量科学数据设计的文件格式及相应库文件，仅在模型的 save/load 函数中使用。

这些使用 conda（优先）或者 pip 命令安装即可，conda、pip 的用法详见附录 A。

### 1. 安装 CUDA（非必需）

可以安装 CUDA 支持 GPU 模式。

CUDA Toolkit 是 NVIDIA 公司面向 GPU 编程提供的基础工具包，也是驱动显卡计算的核心技术工具。从 <https://developer.nvidia.com/cuda-downloads> 下载并安装 CUDA，从 <http://www.nvidia.com/object/mac-driver-archive.html> 下载并安装最新的 CUDA 独立驱动。

在设置路径的时候注意对应版本号：

```
export PATH=/Developer/NVIDIA/CUDA-7.5/bin:$PATH
export DYLD_LIBRARY_PATH=/Developer/NVIDIA/CUDA-7.5/lib:$DYLD_LIBRARY_PATH
```

## 2. 安装 cuDNN (非必需)

GPU 模式下, 在 Keras 中使用 CNN 时可以考虑安装 cuDNN 加速。

从 <https://developer.nvidia.com/cudnn> 下载 cuDNN 库, 解压出来是名为 cuda 的文件夹, 里面有 bin、include、lib。将三个文件夹复制到安装 CUDA 的地方, 覆盖对应文件夹即可。

## 3. 安装 TensorFlow

推荐基于 conda 安装, 运行下面的命令:

```
conda install tensorflow
```

也可以选择 pip 安装, 或者从 <https://github.com/tensorflow/tensorflow> 官网下载源码手动安装。

## 4. 安装 Keras

从 <https://github.com/fchollet/keras> 下载 Keras 源码, 用 cd 命令切换到 Keras 的文件夹中, 并运行下面的安装命令:

```
python setup.py install
```

也可以使用 pip 来安装 Keras:

```
pip install keras -U -pre
```

# B.2 部署 Caffe

下面分别为 Windows 系统用户和 Mac OS 及 Linux 系统用户介绍 Caffe 环境的部署。

## 1. Windows

从 <https://github.com/Microsoft/caffe/releases> 直接下载编译好的包使用, 如果不成功, 再考虑源码安装。

Windows 用户通过源码安装 Caffe 需要准备的工具库如下:

- Visual Studio 2013 or 2015 (首先安装)
- CMake3.4 及以上。

- CUDA (可选)。
- cuDNN (可选)。

大致安装流程和 Mac OS 及 Linux 平台类似, 用 VS 修改配置、编译, 等等。下面主要以 Mac OS & Linux 平台介绍整个安装流程, 这里不再重复介绍流程。

## 2. Mac OS & Linux

Mac OS 和 Linux 整体安装流程相似。

### (1) 安装 CUDA、cuDNN (非必需)

GPU 模式依赖的类库, 见上文。使 cuDNN 生效需要安装好 Caffe 之后, 在 Makefile.config 文件中取消 USE\_CUDNN := 1 的注释。

### (2) 安装 BLAS-Intel MKL(仅针对 Mac 用户)

BLAS 是基础线性代数程序集, Mac 系统自带的 BLAS 库存在不稳定问题, 推荐用 OpenBLAS, 运算矩阵更快, 安装也简单。

```
brew install openblas
```

也可以使用 Intel MKL 库。在校大学生可以使用学校邮箱在 <https://software.intel.com/en-us/qualify-for-free-software/student> 申请 Intel Parallel XE 2015 安装包。

### (3) 安装依赖项

Caffe 的主要依赖包如下。

- Boost C++库。
- Leveldb。
- Lmdb。
- GLog 日志库。
- GFlags 命令行参数库。
- Protobuff (一种数据标准库)。
- HDF5 (一种为存储和处理大容量科学数据设计的文件格式及相应库文件)。
- Snappy 压缩程序库。
- OpenCV 图像处理库。
- szip 压缩库。
- boost-python C++和 Python 语言之间的交互库。



```
brew install -vd snappy leveldb lmdb gflags glog szip
brew tap homebrew/science
brew install wget hdf5 opencv protobuf boost boost-python
```

#### (4) Makefile

下载 Caffe 源码，切换到 Caffe 目录，准备编译配置文件：

```
git clone https://github.com/bvlc/caffe.git
cd caffe/
cp Makefile.config.example Makefile.config
```

使用 CPU 模式打开注释：

```
CPU_ONLY := 1
```

修改 Makefile.config 中关于 BLAS 的配置：

```
# MKL 库 BLAS:=MKL
BLAS := open
BLAS_INCLUDE := /usr/local/Cellar/openblas/0.2.19/include
BLAS_LIB := /usr/local/Cellar/openblas/0.2.19/lib
```

根据自己的 Python 环境修改剩余配置，包括运行命令和 include、lib 库等。Makefile.config 中有推荐对应 pip、brew、anaconda 的配置。以 anaconda 环境为例：

```
ANACONDA_HOME := $(HOME)/anaconda2
PYTHON_INCLUDE := $(ANACONDA_HOME)/include/python2.7 \
    $(ANACONDA_HOME)/lib/python2.7/site-packages/numpy/core/include \
PYTHON_LIB := $(ANACONDA_HOME)/lib
```

仔细看一遍其他配置选项，按个人情况修改。

#### (5) 编译

```
make clean
make all
```

#### (6) 测试

```
make runtest
make pytest
make pycaffe
```

测试程序运行过程中可能会发现缺失一些类库，如 pydot、graphviz 算，使用 conda 或者 brew 安装即可。

### (7) PATH

测试通过后，将编译好的 Caffe 目录放到环境变量中：

```
export PYTHONPATH=<你的 Caffe 安装目录>/python:$PYTHONPATH
```

## 3. 补充

在不同环境下反馈的 Caffe 安装问题较多，大部分集中在编译环节，对于个别存在编译兼容问题的系统，可以考虑安装 cmake 编译。如果还存在问题，则可以考虑牺牲性能，建立 Docker 等虚拟环境运行 Caffe，具体请自行查阅相关资料，这里略过。

目前，Facebook 开源了应用在自家平台上的深度学习框架：Caffe2，优化了移动设备部署和大规模机器部署。同时，官方还提供了从 Caffe 到 Caffe2 迁移的教程，感兴趣的读者可以自行研究。

- Caffe2 的首页：<http://caffe2.ai/>。
- GitHub 地址：<https://github.com/caffe2/caffe2>。

# 随书代码运行环境部署

本书随书代码可以在 <https://github.com/bbfamily/abu> 中下载并使用，如下载地址有变动，请关注公众号 abu\_quant 获取最新地址。随书代码在 IPython Notebook 中运行，运行本书所有示例代码分三步：

- (1) 部署好相关环境。
- (2) 下载随书代码。
- (3) 在终端中，使用 `cd` 命令切换到下载目录中，输入 `jupyter notebook`。

Caffe 相关代码不在 Notebook 中，见相关文件。

下面说明运行环境。

IPython 作为 Python 解释器的增强工具，以非常快捷的代码提示功能而受到广泛的喜好，IPython 有多种环境，本书使用的是 Notebook 环境，其他环境如 Shell 和 Qt 这里不做介绍。

## (1) 安装

Anaconda 默认集成了 IPython 和 Notebook，如果需要单独安装则可以使用 `pip`：

```
pip install ipython jupyter
```

## (2) 启动

启动 Notebook，在 Shell 中键入：

```
jupyter notebook
```

默认配置下会启动电脑的默认浏览器打开 <http://localhost:8888/tree>，显示出当前工作目录下的所有文件信息。如图 C-1 所示，



图 C-1 启动 Jupyter Notebook

### (3) 使用说明

Notebook 中每一个可编辑项都是一个 cell，cell 有 Markdown、code (Python) 等类型。

Notebook 支持 Markdown，让文档的编辑简单高效，即把代码的编写和文档的编写无缝地进行衔接，通过简单的 Markdown 编写文档方式，使开发者不必浪费过多的时间在文档格式、排版、布局等问题上 (Markdown 的详细使用请读者自行查阅相关资料)。如图 C-2 所示，新建一个 ipynb 文件。

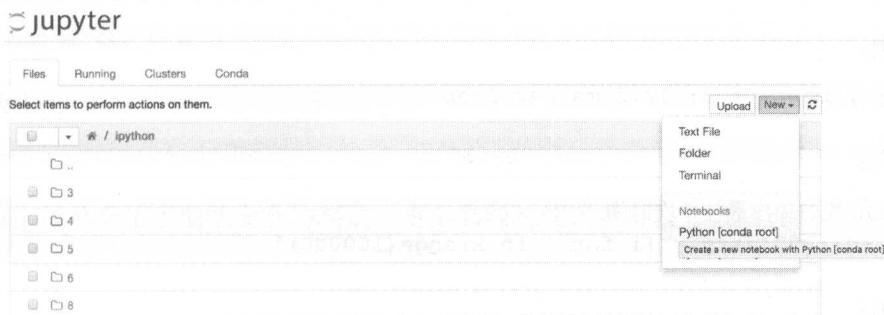


图 C-2 新建

Notebook 的最大优点在于它的交互方式，既可以执行 Python 命令，也可以编辑 Markdown。初次接触 Notebook 就被这种超强的交互方式所吸引，你可以将零散的思路一点一点变成代码，验证代码的有效性，以及更多的实验特性。如图 C-3 所示。

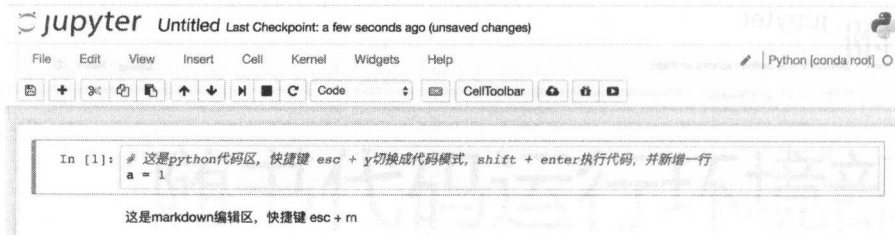


图 C-3 切换 cell 编辑模式

同时, Notebook 提供了一些特殊的功能, 称之为魔法命令 (Magic)。

#### (4) Magic 命令

Magic 命令有两种执行方式: 以%开始的命令被称为行命令, 它只对单行有效; 以%%开头的为单元命令, 它放在单元的第一行, 对整个单元有效。

输入:

```
# 输出 CPU 的执行时间
%time array = [i for i in xrange(100000)]
```

输出:

```
CPU times: user 23.2 ms, sys: 7.14 ms, total: 30.3 ms
Wall time: 26.6 ms
```

输入:

```
# 执行 100 次, 输出最佳单次时间, 用于分析性能十分有效
%timeit array = [i for i in xrange(100000)]
```

输出:

```
100 loops, best of 3: 12.2 ms per loop
```

输入:

```
# 执行 1000 次, 输出最佳单次时间
%timeit -n 1000 array = [i for i in xrange(100000)]
```

输出:

```
1000 loops, best of 3: 12.2 ms per loop
```

- %%time: 针对整个 cell 生效, 计算性能效率。
- %matplotlib inline: 在 Notebook 的 cell 中将可视化图片以网页内嵌的形式展示。



使用示例：

```
%%time
%matplotlib inline # 让Matplotlib 绘图嵌入到 Notebook 中
import matplotlib.pyplot as plt
import math

a = range(0, 360+1) # 度数
x = map(lambda x:math.pi*x/180.0, a)# 弧度
plt.plot(x,map(lambda x:math.sin(x), x))
plt.show()
```

输出如图 C-4 所示。

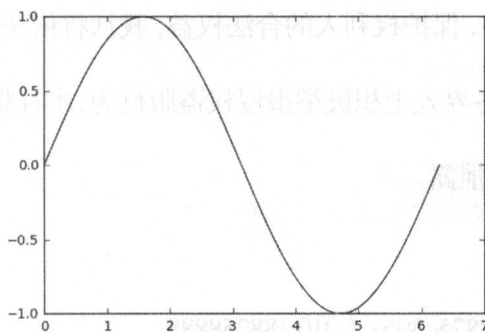


图 C-4 正弦函数

```
CPU times: user 237 ms, sys: 8.75 ms, total: 245 ms
Wall time: 245 ms
```

还支持一些简单的 Shell 命令，如：

```
!open non-exist.txt
```

输出：

```
The file /Users/baidu/maxmon/ml_book/notebook/non-exist.txt does not exist.
```

以上简单介绍了魔法命令的用法，更多高级玩法读者可自行查阅资料。

---

投稿联络：安娜

微信&QQ：80303489

邮箱：anna@phei.com.cn

---



# 机器学习之路

Caffe、Keras、scikit-learn实战



机器学习需要一条脱离过高理论门槛的学习之路。

本书不谈高深晦涩的数字理论，也不谈复杂的算法代码实现，本书只通过朴实近人的方式谈谈如何理解机器学习。

- ▶ 比如通过小红帽采蘑菇的故事对比人类和计算机学习的差异
- ▶ 通过预测房价、量化交易股票来展示如何使用机器学习技术解决生活中的实际问题
- ▶ 从理解大脑到理解深度学习，探索使用工程技术模拟人脑的智慧
- ▶ 最后体验使用深度学习技术有效识别狗照、创作艺术画时的神奇魅力

人生的精力有限，让我们关注那些力所能及的重要事情。



博文视点Broadview



@博文视点Broadview



责任编辑：安娜  
封面设计：吴海燕

上架建议：机器学习

ISBN 978-7-121-32160-3



9 787121 321603 >

定价：79.00元